2006-03

Standards interoperability application of contemporary software assurance standards to the evolution of legacy software

Meacham, Desmond J.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/2985

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

STANDARDS INTEROPERABILITY: APPLICATION OF CONTEMPORARY SOFTWARE SAFETY ASSURANCE STANDARDS TO THE EVOLUTION OF LEGACY SOFTWARE

by

Desmond J. Meacham

March 2006

Thesis Advisor:	James B. Michael
Second Reader:	Jeffrey M. Voas

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2006	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE: Standards Interoperability: Application of Contemporary Software Assurance Standards to the Evolution of Legacy Software	**5. FUNDING NUMBERS**
6. AUTHOR(S) Desmond J. Meacham	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

This thesis addresses software evolution from the perspective of standards interoperability. We address the issue of how to apply contemporary software safety assurance standards to legacy safety-critical systems, with the aim of re-certifying the legacy systems to the contemporary standards. The application of RTCA DO-178B 'Software Considerations in Airborne Systems and Equipment Certification' to modified legacy software is the primary focus of this thesis. We present a model to capture the relationships between pre- and post-modification software and standards. The proposed formal model is then applied to the requirements for RTCA DO-178B and MIL-STD-498 as representative examples of contemporary and legacy software standards. The results provide guidance on how to achieve airworthiness certification for modified legacy software, whilst maximizing the use of software products from the previous development.

14. SUBJECT TERMS Airworthiness, Legacy Software, MIL-STD-498, RTCA DO-178B, Software Assurance, Software Certification, Software Evolution, Standards Interoperability, Software Reuse, Abstract Algebra	**15. NUMBER OF PAGES** 99
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

STANDARDS INTEROPERABILITY: APPLICATION OF CONTEMPORARY SOFTWARE ASSURANCE STANDARDS TO THE EVOLUTION OF LEGACY SOFTWARE

Desmond J. Meacham
Flight Lieutenant, Royal Australian Air Force
B.ENG., University of Newcastle, Australia, 2000

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 2006

Author: Desmond J. Meacham

Approved by: Prof. James B. Michael
Thesis Advisor

Dr. Jeffrey M. Voas, SAIC Inc.
Second Reader

Prof. Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis addresses software evolution from the perspective of standards interoperability. We address the issue of how to apply contemporary software safety assurance standards to legacy safety-critical systems, with the aim of recertifying the legacy systems to the contemporary standards. The application of RTCA DO-178B 'Software Considerations in Airborne Systems and Equipment Certification' to modified legacy software is the primary focus of this thesis. We present a model to capture the relationships between pre- and post-modification software and standards. The proposed formal model is then applied to the requirements for RTCA DO-178B and MIL-STD-498 as representative examples of contemporary and legacy software standards. The results provide guidance on how to achieve airworthiness certification for modified legacy software, whilst maximizing the use of software products from the previous development.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF EQUATIONS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ABBREVIATIONS, ACRONYMS, AND SYMBOLS

Term	Definition
ADF	Australian Defence Force
ANSI	American National Standards Institute
ARP	Aerospace Recommended Practice
CM	Configuration Management
COM	Computer Operation Manual
COTS	Commercial off the shelf
CPM	Computer Programming Manual
CSCI	Computer Software Configuration Item
DBDD	Database Design Description
DEF STAN	Defence Standard (UK)
DID	Data Item Description
ECSS	European Cooperation for Space Standardization
EUROCAE	European Organisation for Civil Aviation Equipment
F	Fighter
F/A	Fighter/Attack
FAA	Federal Aviation Administration
FSM	Firmware Support Manual
HTH	Hierarchical Typed Hypergraph
IAW	In accordance with
IDD	Interface Design Description
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronic Engineers
IRS	Interface Requirements Specification
ISO	International Organization for Standardization
JHMCS	Joint Helmet Mounted Cueing System
JSSSC	Joint Software System Safety Committee
MC	Mission Computer
MIL-HDBK	Military Handbook (U.S.)

Term	Definition
MIL-STD	Military Standard (U.S.)
MISRA	Motor Industry Software Reliability Association
MoD	Ministry of Defence
NASA	National Aeronautics and Space Administration
OCD	Operational Concept Description
OFP	Operational Flight Programs
PSAC	Plan for Software Aspects of Certification
RAAF	Royal Australian Air Force
ROI	Return On Investment
RTCA	Radio Technical Commission for Aeronautics
S/W, Sw	Software
SAE	Society of Automotive Engineers
SCMP	Software Configuration Management Plan
SCOM	Software Center Operator Manual
SCS	Software Configuration Set
SDD	Software Design Description
SDP	Software Development Plan
SIOM	Software Input/Output Manual
SIP	Software Installation Plan
SLOC	Source lines of code
SMP	Stores Management Processor
SPS	Software Product Specification
SQA	Software Quality Assurance
SQAP	Software Quality Assurance Plan
SRS	Software Requirements Specification
SSDD	System/Subsystem Design Description
SSS	System/Subsystem Specification
Std	Standard
STD	Software Test Description
STP	Software Test Plan
STR	Software Test Report

Term	Definition
STrP	Software Transition Plan
SUM	Software User Manual
SVD	Software Version Description
SVP	Software Verification Plan (DO-178B)
TAR	Technical Airworthiness Regulator (ADF)
TR	Technical Report
UK	United Kingdom of Great Britain and Northern Ireland
UML-RT	Unified Modeling Language for Real-Time
U.S.	United States of America
USMC	United States Marine Corps
USN	United States Navy
A (Alpha)	Set of all possible symbols defined for a given algebra
A_x	Superset of software products for version X
B_S, C_S	Subset of software products of A_x
R_x	Relationship 'x' (between software and/or standard)
S_x	Software standard 'x'
X^n	nth version of software X
α_r	A single software product
ϕ	Morphism function
ω_S	A single software development activity
Ω	Set of all possible operators defined for a given algebra
$\rightarrow$	Produces
$\mathbb{R}$	Set of all real numbers
$\mathbb{R}^+$	Set of all positive real numbers
$\varnothing$	Null set
$\bigcup$	Set union
$\subset$	Subset
$\bigcup_{s=1}^{n}$	Union of n sets

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF SAFETY AND SOFTWARE STANDARDS AND GUIDELINES

Designation	Status	Title
ANSI/IEEE Std 730	2002	Standard for Software Quality Assurance Plans
MoD Def Stan 00-54	Superseded	Requirements for Safety Related Electronic Hardware in Defence Equipment (UK)
MoD Def Stan 00-55	Superseded	Requirements for Safety Related Software in Defence Equipment (UK)
MoD Def Stan 00-56	Interim Issue 3	Safety Management Requirements for Defence Systems (UK)
MoD Def Stan 00-58	Superseded	HAZOP Studies on Systems Containing Programmable Electronics (UK)
DEF(AUST) 5679	1998	The Procurement of Computer-Based Safety Critical Systems
DI(AF) AAP 7001.054	2004	Airworthiness Design Requirements Manual
EN 50126	1999	Railway Applications – The Specification and Demonstration of Reliability, Availability, Maintainability and Safety
EN 50128	2001	Software for Railway Control and Protection Systems
FAA Order 8110.49	2003	Software Approval Guidelines (U.S.)
H ProgSäkE	2001	Handbook for Software in Safety Critical Applications (Sweden)
H SystSäkE		System Safety Activities for Defense Systems (Sweden)
IEC 60880	1986	Software for Computers in the Safety Systems of Nuclear Power Stations
IEC 60880-2	2000	Software for Computers Important to Safety for Nuclear Power Plants - Part 2: Software Aspects of Defence Against Common Cause Failures, use of Software Tools and of Pre-Developed Software
IEC 61508	1998	Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems
IEEE Std 829	1998	Standard for Software Test Documentation
IEEE Std 1012	2004	Standard for Software Verification and Validation
IEEE Std 1028	1997	Standard for Software Reviews

Designation	Status	Title
IEEE Std 1228	1994	Standard for Software Safety Plans
IEEE Std 1298	1992	Software Quality Management System
IEEE/EIA 12207	1996/97	Industry Implementation of ISO/IEC 12207-1995 Information Technology – Software Life Cycle Processes
ISO 9000-3	1997	Quality Management and Quality Assurance Standards – Guidelines for the Application of ISO 9001:1994 to the Development, Supply, and Maintenance of Computer Software
ISO/IEC 15026	1998	Information technology – System and software integrity levels
ISO/IEC TR 15942	2000	Information Technology – Programming languages – Guide for the Use of the Ada Programming Language in High Integrity Systems
JSSSC SSSH	1999	Software System Safety Handbook (U.S.)
MIL-HDBK-286		A Guide for DOD-STD-2168 (U.S.)
MIL-STD-2167A	Cancelled	Defense System Software Development (U.S.)
MIL-STD-498	1994	Software Development and Documentation (U.S.)
MIL-STD-882C	Superseded	System Safety Program Requirements (U.S.)
MIL-STD-882D	2000	Standard Practice for System Safety (U.S.)
RTCA DO-178B EUROCAE ED-12B	1992	Software Considerations in Airborne Systems and Equipment Certification
RTCA DO-248B EUROCAE ED-94B	2001	Final Report for Clarification of DO-178B "Software Considerations in Airborne Systems and Equipment Certification"
SAE ARP 4754	1996	Certification Considerations for Highly-Integrated or Complex Aircraft Systems
SAE ARP 4761	1996	Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment
UL 1998	1998	Standard for Safety-Related Software (U.S.)

GLOSSARY

Term	Definition	Source
Arity	The number of arguments a function or operator takes.	computing-dictionary
Quality Assurance	(1) A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements. (2) A set of activities designed to evaluate the process by which products are developed or manufactured.	IEEE610.12
Safety Assurance	(1) The degree of confidence required in the correctness of a particular process or tool. (2) The planned and systematic actions necessary to provide adequate confidence and evidence that a product or process satisfies given requirements.	(1) DEF STAN 00-55 (2) DO-178B
Software Assurance	Software assurance is the planned and systematic set of activities that ensures that software processes and products conform to requirements, standards, and procedures. ‘Processes’ include all of the activities involved in designing, developing, enhancing, and maintaining software; ‘products’ include the software, associated data, its documentation, and all supporting and reporting paperwork.	AAP 7001.054
Software Quality	The ability of software to satisfy its specified requirements.	MIL-STD-498
System Safety	The application of engineering and management principles, criteria, and techniques to achieve acceptable mishap risk, within the constraints of operational effectiveness and suitability, time, and cost, throughout all phases of the system life cycle.	MIL-STD-882D

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I wish to express my sincere gratitude to the Royal Australian Air Force for giving me the opportunity to pursue the Master of Science in Software Engineering at the Naval Postgraduate School. In a climate of scarce resources, the opportunity to pursue educational opportunities in a foreign country is rare. I am also grateful that the United States Navy and the Naval Postgraduate School have the wisdom to offer this degree program, seeing value in this program of study when other institutions do not.

I must express my gratitude to Professor Michael, my academic associate and thesis advisor, for sharing his knowledge and experience with me. He was always willing to offer his insight and advice on this thesis and other coursework, despite the considerable constraints on his time. He also deserves special recognition for regularly stepping-in to fill-the-gap when a course required an instructor. To him, and the other members of the Software Engineering Department, I owe a great debt.

This thesis is dedicated to my wife who resolutely endured the trials of being a study-widow, again, and provided the unfailing support that permitted me to focus during the duration of the course.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. EVOLUTIONARY DEVELOPMENT OF AIRCRAFT AND AEROSPACE SOFTWARE

There are many reasons for software evolution. Seacord, Plakosh and Lewis [1] identify three categories of evolution: (i) maintenance, (ii) modernization and (iii) replacement. They describe maintenance as small changes that are typically corrections to software faults or minor enhancements. Modernization involves major changes to a system, but which preserve a significant amount of the old system. Modernization may take the form of *retargeting* old software to a new hardware platform; *revamping* the human machine interface to improve usability; *component substitution*, such as with alternate commercial products; *source code translation* to new versions of the same language or different languages to that previously used; *code reduction* to remove unused functionality or re-factor remaining functionality; and *functional transformation* to achieve structural improvement. Replacement involves adopting a completely new design for a system when the old system cannot be modernized in an effective manner.

Seacord, Plakosh and Lewis go on to identify *complexity*, *software technology and engineering processes*, *risk*, *commercial components*, and changed *business objectives* as challenges to modernization. Leveson [2] identifies the appearance of *new hazards*, an increased *exposure* to software-intensive systems, greater amounts of *energy* being monitored or controlled by software, and an increased *reliance* on software for monitoring and control of systems as further challenges to software maintenance. Additional challenges experienced in the military domain include the need for high degrees of *inter-operability* with external systems; national and international rather than personal or commercial *security* concerns; and a harsh *operating environment* (i.e., combat) in which the system must continue to operate.

Military aerospace systems are examples of software-intensive systems that exhibit many of the aforementioned challenges. Such systems are costly to produce and take many years to develop. For example, consider the following timeline for the Boeing (McDonnell Douglas) F/A-18A/B/C/D: [3,4,5,6]

1970s	Predecessor design (Northrop YF-17) proposed as an Air Combat Fighter for the United States Air Force. (F-16 chosen instead).
1975	Modified design (F-18) accepted as a Naval Air Combat Fighter for the United States Navy (USN) and Marine Corps (USMC).
1970/1980s	Variant designs for Attack (A-18) and Trainer (TF-18) versions developed but eventually merged to become the F/A-18A single-seat and F/A-18B dual seat versions.
1978 & 1979	First flights of an F/A-18A & B respectively.
1980-1988	F/A-18A & B aircraft delivered to USN and USMC.
1982-1988	CF-18A & B aircraft delivered to Canadian Forces – Air Command.
1984-1990	AF/A-18A & B aircraft delivered to Royal Australian Air Force (RAAF). [7]
1986-1990	EF-18A & B aircraft delivered to Spanish Air Force.
1987-2000	F/A-18C & D delivered to USN and USMC.
1991-1993	F/A-18C & D aircraft delivered to Kuwait Air Force.
1992-2008	Upgrade of Spanish Air Force EF-18A & B aircraft.
1995-2000	F-18C & D aircraft delivered to Finnish Air Force. [8]
1996-1999	F/A-18C & D aircraft delivered to Swiss Air Force.
1997	Delivery of F/A-18D aircraft to Royal Malaysian Air Force.
1997	Merger of the Boeing and McDonnell Douglas companies.
1999 onwards	Upgrade of USN and USMC F/A-18A, B, C & D aircraft.
2002-2010	Upgrade of RAAF F/A-18A & B aircraft. [7]
2002-2009	Upgrade of Canadian Forces – Air Command CF-18A & B aircraft. [9]
2004-2008	Upgrade of Swiss Air Force F/A-18C & D aircraft.
2007-2014	Upgrade of Finnish Air Force F-18C & D aircraft. [8]
2015	Planned withdrawal of RAAF AF/A-18 aircraft. [10]
2017-2020	Planned withdrawal of the Canadian Forces – Air Command CF-18. [9]
2020	Planned withdrawal of Spanish Air Force EF/A-18 aircraft
2025	Planned withdrawal of Finnish Air Force F-18 aircraft. [8]

Military aircraft are not alone in their long life and continual upgrade. After three years of initial design work on the original design, the Boeing 747-100 design was accepted in 1966 and entered service in 1970. In the forty years since then, four other significant variants and thirteen minor variants have been or are being built [11]. The delivery of the latest 747-8 aircraft is conjectured to last twenty years from a planned service date of 2009. It is conceivable that these aircraft will be flown for twenty years beyond final delivery. While the last design will be substantially different from the first, this represents around eighty years of evolution.

The F/A-18 timeline above only includes four major variants of the F/A-18 without mention of the different configurations of equipment and software for the eight nations that utilize this aircraft. Nor does it include the three F/A-18E/F/G variants which some consider to be substantially different from the earlier variants of the aircraft. The timeline above reveals an approximately thirty year life to date and around fifty years total for development, maintenance, and modernization from the conception of the F/A-18 to its planned final disposal. Two dominant reasons for changes to aircraft are new mission requirements and technology improvements, such as the addition of an attack role as well as the air combat role for the F/A-18, and the availability of new equipment such as the Joint Helmet Mounted Cueing System (JHMCS) or new weapons such as the AIM-9X air-to-air missile. Given the long development time and service history of aircraft, many changes to an aircraft design can be expected over its lifespan. Furthermore, the considerable amount of previous expenditure on an existing aircraft invites modernization of the existing platform before purchasing a new aircraft. Unit costs of F/A-18A & B aircraft have been reported between USD28-35 million.

A critical enabler for variants and upgrades of all aircraft is software. The collection of Operational Flight Programs (OFP) in the many different processors of the F/A-18 is collectively called a Software Configuration Set (SCS). Some of the SCS that have been developed or are currently in development or planning for the F/A-18A/B/C/D include 89C, 91C, 92A, 09C, 10A, 11C, 12A, 13C, 15C, 17C, 18E, 19C, 21C, 23C, 25C. Some of these SCS were integral to the hardware upgrade programs that were listed in the timeline. However, most of them are new versions of an SCS for the same target platform. In addition to this list of SCS, there are also different versions of some of the above SCS for different international customers, for example, 15C for the USN and USMC but 15CA for the RAAF; and at least two countries outside the United States (U.S.) are both maintaining and modernizing the SCS that they receive to meet their own unique requirements and priorities. The 15C SCS is a recent software development that demonstrates many of the challenges to software modernization. The 15C SCS was delivered in 2001 after four years of development. This SCS started out with seventy-five high-level Statements of Requirements to be completed in three builds and ended with 134 Statements of Requirements delivered over four builds. The final product integrated three new weapons and five new major avionics systems. The 15C SCS has over ten million source lines of code (SLOC), across more than forty processors and uses twelve languages in the aircraft with a further two in the development environment which itself has four million SLOC [12]. Several computer processors in the F/A-18 required upgrading, forcing *retargeting* of the OFP to run on them. Replacement color displays and the integration of the JHMCS have enabled a *revamping* of parts of the human machine interface.

B. SAFETY-CRITICAL SOFTWARE CERTIFICATION

An essential part of developing safety-related aerospace software that is expected to be operationally fielded is to comply with the airworthiness design requirements specified by the certification authority. Examples of airworthiness certification authorities include the Federal Aviation Administration (FAA) for civilian aviation in the U.S., the Australian Defence Force (ADF) for military aviation in Australia, and the Naval Air Systems Command for USN and USMC aviation. Airworthiness design requirements address the acceptable level of confidence required in the safety of all parts of the aircraft system: hardware, software and the human operators (sometimes referred to as "skinware"). *Software* system safety requirements address those parts of a system for which software is identified as the *source* of, *detector* of, or *means of containing* a system fault, regardless of the locus of the fault. "Certification is normally based on the use of some form of standard…" [13]. The current ADF preference for a software assurance standard for the development of safety-related aerospace software is the Radio Technical Commission for Aeronautics (RTCA) DO-178B 'Software Considerations in Airborne Systems and Equipment Certification' [14].

Obtaining an appropriate return on investment (ROI) is an understandable expectation for the acquirer of any system. The considerable amount of resources it takes to develop a large aerospace system necessitates a relatively lengthy time of system operation in order to recoup this investment. The system will change over time to account for one or more of the following: corrections to design faults or implementation flaws in the previous system, adaptations of existing system functions to accommodate changes in environment or operations, or completely new features that meet previously infeasible or unimagined requirements. One element for consideration when *modifying* software is to maintain at least an equivalent level of assurance as that for the initial development. However, legacy software previously developed to standards such as DOD-STD-2167A or MIL-STD-498 may not have included adequate provision for software assurance that would meet today's standards. As such, a desirable goal during software modification is to upgrade the previous certification basis by addressing the objectives of a contemporary software assurance standard such as DO-178B [15]. Applying DO-178B guidance to the development of new software requires considerable effort at safety levels of higher integrity, but is relatively straightforward when compared with the re-certification of legacy software to DO-178B that did not have the DO-178B guidelines applied during the previous software development.

Developers may choose to improve their software development processes when modifying legacy software in order to achieve certification to a new software assurance standard. When an applicant seeks re/certification, the certification authority takes into account whether the evidence provided by the software development team sufficiently addresses the software assurance objectives, activities and considerations. The level of confidence one has that the modified software deserves certification against a new software assurance standard should increase each time the legacy software is further modified and re-certification is subsequently achieved, *ceteris paribus*. This approach to certification of evolving software is the current practice of the ADF.

However, re-certification of software to a new software assurance standard may be built upon legacy software that in some fundamental way does not warrant certification to a new standard of level. Whilst the improved processes applied during modification address the modification itself and any identified interfaces to non-modified software, the processes may not address the fundamental system safety properties of that part of the underlying legacy software that is not addressed in the modification or is not identified as interfacing the modified areas. This approach to re-certifying legacy software raises the following question: Could legacy software be fundamentally flawed in areas that are left unmodified during software evolution and result in unwarranted certification of software to a new software assurance standard?

C. UNDERLYING CONCEPTS AND DEFINITIONS

1. Legacy Software

For the purposes of this thesis, the term *legacy software* means software that has been previously developed and is subject to modification, that is, both maintenance and modernization. More specifically, it is software that has been developed to a defined standard, or through a defined process, so that the software has a known pedigree, but a pedigree that is not currently desirable. It is recognized that this is a narrow definition and does not, amongst other scenarios, include the reuse of software in a new application without first being modified.

2. Software Safety within System Safety

MIL-STD-882D has the following definition for safety:

> Freedom from those conditions that can cause death, injury, occupational illness, damage to or loss of equipment or property, or damage to the environment. [16]

Leveson defines safety in a manner consistent with this absolute point of view as "freedom from accidents or losses," but recognizes that this is not achievable for real-world systems, especially those systems that are complex. Absolute safety is, however, the goal that should be the starting point from which judgments about acceptable levels of mishap risk are made. Leveson goes on to make the case that safety is a system property that has a contribution from software whenever software is involved in a system. The definition for system safety from MIL-STD-882D is:

> The application of engineering and management principles, criteria and techniques to achieve acceptable mishap risk within the constraints of operational effectiveness, time and cost throughout the system's life cycle.

This definition and its nearly identical predecessor in MIL-STD-882C start out with a provisional view of system safety that is a function of system effectiveness and development schedule and cost.

Roland and Moriarty [17] state that the following as the concept for system safety:

> … involves a planned, discipline, systematically organized, and before-the-fact process characterized as the identify-analyze-control method of safety.

In both of the preceding definitions, system safety is assumed to be reliant on the process by which a system is developed; that is, system safety does not simply happen by chance, but is instead part of system design. However, just having a development process does not guarantee that a system will be safe (whatever your definition of safety). The process must be suitable, rigorous, complete and actually used to develop a product that can be regarded as relatively safe with some degree of confidence in the assertion of safety.

The application of system safety engineering focuses on the early identification and analysis of hazards which in turn permits the system developer to mitigate them through system design. This is the preferred method of treating systems hazards. The alternative is late identification of hazards which forces either the treatment of hazards through the less desirable procedural and training mitigation measures, or the costly rework (in time and money) to incorporate design mitigation measures.

Software is an abstraction and as such, it has no substance and cannot directly harm people, property or the environment. However, software can be responsible for the loss. The opportunity for software to contribute to loss is enabled through its use by system developers to sense and control physical components within a system and its environment; the physical components can release energy that can harm someone or something. Safety of software is just one of a system's properties, and is dependent on the software, hardware, operator and external factors. Leveson provides the following definition of software system safety that is consistent with this perspective: “Software System Safety implies that the software will execute within a system context without contributing to hazards” [2].

3. Software Assurance

The ADF Airworthiness Design Requirements Manual describes software assurance in the following terms:

> Software assurance is the planned and systematic set of activities that ensures that software processes and products conform to requirements, standards, and procedures. ‘Processes’ include all of the activities involved in designing, developing, enhancing, and maintaining software; ‘products’ include the software, associated data, its documentation, and all supporting and reporting paperwork. [14]

The Institute of Electrical and Electronic Engineers (IEEE) Standard Glossary of Software Engineering Terminology does not define software assurance but does have an entry for software quality assurance which refers readers to quality assurance which states:

> (1) A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements.
>
> (2) A set of activities designed to evaluate the process by which products are developed or manufactured. [18]

The National Aeronautics and Space Administration (NASA) does define Software Assurance, and does so in terms from the IEEE Standard Glossary [18], but adds that NASA’s definition includes the disciplines of software quality, safety, reliability, and verification and validation:

> The planned and systematic set of activities that ensure that software life cycle processes and products conform to requirements, standards, and procedures. [19]

NASA elaborates on this definition of software assurance by introducing functional or mission-requirement elements and adds the aspect of oversight to the assurance activity:

> Software assurance is an umbrella risk mitigation strategy for safety and mission assurance of all of NASA's software. The purpose of software assurance is to assure that software products are of high quality and operate safely.
>
> Assure is used when software assurance practitioners make certain that the specified software assurance, management, and engineering activities have been performed by others.

Finally, for plain English definitions of *assurance*, the following are taken from Wiktionary:

> (1) The act of assuring; a declaration tending to inspire full confidence; that which is designed to give confidence.
>
> (2) The state of being assured; firm persuasion; full confidence or trust; freedom from doubt; certainty. [20]

Based on the preceding definitions, one can conclude that the aim of software assurance is to provide confidence that the software complies with all of its requirements (e.g., functional, safety, reliability) and that software assurance is distinct from other software development activities. In some cases software assurance activities require an independent oversight to justify the non-independent claims of software assurance.

4. Mission Hazards

An additional consideration for military systems is mission-worthiness. If part or all of a combat military system's (e.g., tank, ship, aircraft) self-defense mechanisms is inoperable, the system will be subject to additional mission hazards that are a part of the combat environment, for instance, loss of lives and equipment through destruction or capture. Unacceptable consequential losses may also be incurred by either of the combatants if a military system's offensive capability is faulty or inoperable, such as failure to destroy an enemy may result in subsequent loss of friendly forces, or inaccuracies in targeting may increase collateral damage and loss. These examples do not fit the contemporary view of safety hazards, but rather operational or performance failures that have hazards as an indirect consequence. However, due to the dire consequences of some operational or performance failures in a combat environment, application of the techniques that assure safety of software can also be applied to the mission requirements of software-intensive systems.

THIS PAGE INTENTIONALLY LEFT BLANK

II. SAFETY-CRITICAL SOFTWARE STANDARDS

A. EXAMPLES FROM THE AEROSPACE DOMAIN

The focus of this thesis is the application of DO-178B as it is the preferred standard for software assurance for safety-related airborne software in the ADF. This is not to say that alternative standards do not exist or are unacceptable. In the aerospace domain, software-related standards in use include:

RTCA DO-178B 'Software Considerations in Airborne Systems and Equipment Certification' in conjunction with an acceptable system/software safety standard for civilian aviation in the U.S.

MIL-STD-498 'Software Development and Documentation' in conjunction with MIL-STD-882D 'Standard Practice for System Safety' for military aviation in the U.S.

DEF STAN 00-55 'Requirements for Safety Related Software in Defence Equipment'[1] produced by the United Kingdom Ministry of Defence

NASA-STD-8739.8 'Software Assurance Standard' for the development of aerospace software within NASA

ECSS Q-80B 'Software Product Assurance' by the European Space Agency

B. EXAMPLES FROM OTHER DOMAINS

Aerospace software is just one safety-critical domain that requires standards for software development and/or assurance. Other domains and the standards proposed for them include:

EN 50128 'Software for railway control and protection systems' and IEC 62279 'Software for railway control and protection systems' for the development of software in the railway domain

IEC 60880 'Software for computers in safety systems of nuclear power stations' and IEC 62138 'Software aspects for computer-based systems performing category B or C functions' for application to nuclear power station software

IEC 60601-1-4 'General requirements for safety - Collateral Standard: Programmable electrical medical systems' for software in the medical equipment domain

[1] Now obsolete and superceded by DEF STAN 00-56 'Safety Management Requirements for Defence Systems'.

ISO/TR 15497 'Road vehicles – Development guidelines for vehicle based software' and the Motor Industry Software Reliability Association (MISRA) Report 2 'Integrity' for software in the road vehicle domain

The following list of standards has been proposed for general use, rather than being identified for application in a single domain:

IEC 61508 'Functional safety of electrical/electronic/programmable electronic safety-related systems'

ISO/IEC 12207 'Information technology – Software life cycle processes'

ISO/IEC 9126 'Software engineering – Product quality'

ISO/IEC 14598 'Software engineering – Product evaluation'

ISO/IEC 90003 'Software engineering – Guidelines for the application of ISO 9001:2000 to computer software'

C. RTCA DO-178B – SOFTWARE CONSIDERATIONS IN AIRBORNE SYSTEMS AND EQUIPMENT CERTIFICATION

1. DO-178B Fault Condition Categories and Safety Levels for Software

DO-178B was developed in collaboration with the European Organisation for Civil Aviation Equipment (EUROCAE) which published the document as ED-12B with the same title. The RTCA is not an officially sanctioned authority, and as such, DO-178B is not mandatory for use within the U.S.. However, DO-178B is highly regarded within the aviation community and is the preferred software assurance standard for safety-related airborne software by the FAA and ADF. DO-178B pertains to the assurance and certification of all software requirements, not just safety-related software requirements, but does make specific mention of the safety-related requirements that are imposed on aerospace software by the system safety process.

Before the guidelines of DO-178B can be applied, a system safety assessment process (not included as part of DO-178B) is used to determine the sources of any safety-related requirements and the failure condition categories associated with them. Two system safety standards that may be used for this purpose are MIL-STD-882C [21] and the SAE ARP4754 [22]. Safety-related requirements are allocated to the various sub-systems during the system safety program. Those requirements that have been allocated to software as the source, the detector or the method of fault containment, are allocated a safety level for the most severe failure condition category associated with that component of software from the following list in Table 1. For example, the flight control sub-system would very probably warrant a failure condition category of catastrophic (as determined by the system safety program) and hence the software component of the flight control subsystem would be developed as safety level A software.

Failure Condition Category	Safety Level
Catastrophic	A
Hazardous/Severe-Major	B
Major	C
Minor	D
No Effect	E[2]

Table 1. RTCA DO-178B Safety Categories and Software Levels

The allocated safety level then determines the following: which subset of activities must be conducted, the degree of rigor to be applied to the activities, whether the assurance of them needs to be conducted independently from development, and the control category for data management of the software products.[3]

2. DO-178B Objectives, Activities, Considerations and Evidence

DO-178B is an objective-based standard that specifies the objectives for three categories of software lifecycle processes; the three categories are planning, development and integral processes. For each process, DO-178B defines:

a. Activities to achieve software life cycle objectives,

b. Design considerations to support software lifecycle objectives, and

c. Evidence that demonstrates that software lifecycle objectives have been achieved.

DO-178B also identifies by safety level what control should be placed on the data items produced and whether an objective and activity should be conducted by parties independent of the software development.

[2] Level E software does not require the application of any DO-178B activities, considerations or evidence.

[3] Use of the term ‘software product’ within this thesis is synonymous with ‘software artifact’ and consistent with the use in [27].

In total, DO-178B lists sixty-six planning, development and integral objectives that are applicable to software assessed as being safety level A. For safety level D software, a subset of only twenty-eight objectives are required. The complete set of objectives are listed in Annex A of this thesis, and grouped as listed in Table 2. Numbers in parentheses indicate the number of objectives that are required to be independently assessed, rather than assessed by the software product developer.

Lifecycle Processes	Safety Level			
	A	B	C	D
Planning	7	7	7	2
Development	7	7	7	7
Verification	40 (22)	39 (11)	32	8
Configuration Management	6	6	6	6
Quality Assurance	3 (3)	3 (3)	2 (2)	2 (2)
Certification Liaison	3	3	3	3
Total	66	65	57	28

Table 2. Breakdown of Objectives by Lifecycle Process and Safety Level

As can be seen, the greatest number of objectives for software safety levels A, B and C are verification, totaling nearly two-thirds of the objectives required for each of these safety levels. The verification activity spans the range of software development activities. Planning, development, configuration management, quality assurance and certification liaison objectives are almost uniformly applied across safety levels A through D.

3. Circumstances for the Application of DO-178B

In addition to being applied to the development of original software, DO-178B may be applied under any of the following circumstances:

a. Software that was previously developed and certified to DO-178B and is to be modified and certified to the same safety level as previously achieved

b. Software that was previously developed for a different aircraft installation and may or may not be subject to modification before seeking certification

c. Software that was previously developed using a different development environment or developed for a different application environment

d. Software that was previously developed to different standards or guidelines, such as commercial-off-the-shelf (COTS) software, a different standard, or to DO-178B but to a different safety level.

A note to section 2.2.3 of DO-178B provides the following advice regarding software modification:

> The applicant may want to consider planned functionality to be added during future developments, as well as potential changes to system requirements allocated to software that may result in a more severe failure condition category and higher software level. It may be desirable to develop the software to a level higher that that determined by the system safety assessment process of the original application, since later development of software life cycle data for substantiating a higher software level application may be difficult.

The use of DO-178B during software evolution is not without its problems. Johnson identified literal interpretation of the current standard and incorrect application of its predecessors, DO-178A and DO-178, as areas of concern.

> Challenges in using DO-178B are already occurring. They include the discovery that previously certified systems didn't necessarily use earlier versions of DO-178 correctly and now result in greater transition issues. Literal interpretation remains a problem. [23]

D. MIL-STD-498 – SOFTWARE DEVELOPMENT AND DOCUMENTATION

This section presents a brief description of the software requirements specified in MIL-STD-498. MIL-STD-498 is chosen as a representative legacy software standard because of the prevalence of military aerospace systems still in operation today that were developed in accordance with (IAW) this standard.

1. Background and Scope of MIL-STD-498

MIL-STD-498 was developed to resolve the objections to MIL-STD-2167A Defense System Software Development and to merge its contents with those of 7935A DoD Automated Information Systems Documentation Standards, thus forming a single *best-of-both* standard that was consistent with other Department of Defense policy and instructions that were released at around the same time of issue of MIL-STD-498.

One of the significant changes to the requirements in MIL-STD-498 was an attempt to be independent of any particular development methodology. The standard provides considerable guidance for application to the grand design, incremental design, and evolutionary design development methodologies as an example of the proposed flexible use of the standard. Another deliberate change from earlier standards was a recognition that software product data could and should be provided in alternative forms to traditional documents, in fact advocating the use of natural work products rather than development of additional documents as evidence of activities and results.

Attempting to merge weapon system software and information system standards led to a document that has a number of requirements that have little or no relevance to airborne software, for instance, a software center operating manual. This situation does not present a problem as MIL-STD-498 repeatedly advises that the requirements of the standard should be tailored to suit the particular software development project. Guidance and instructions for tailoring are provided in each of the general and detailed requirements and in the associated Data Item Descriptions (DID). The standard also makes reference to the general tailoring guidance in MIL-HDBK-248 Acquisition Streamlining.

2. MIL-STD-498 Process and Product Requirements

Approximately thirty general and detailed requirements and approximately twenty-nine types of software products are described in MIL-STD-498 which references the content in the twenty-two accompanying DID for specific information. Development processes that are required by the standard include:

a. Participation in system-level activities (requirements through to testing),

b. Software requirements analysis, design, and implementation,

c. Verification, integration, testing and corrective action,

d. Configuration and risk management, metrics analysis, quality assurance, reviews, audits, and

e. Installation and transition.

Software development products are introduced and briefly described in the standard and more fully described in the accompanying DID. The 22 DID specified by MIL-STD-498 include:

a. Plans for the conduct of software development activities
   1. Software Development Plan
   2. Software Test Plan
   3. Software Installation Plan
   4. Software Transition Plan

b. Specifications of system, software and software
   1. System/Subsystem Specification
   2. Interface Requirements Specification
   3. Software Requirements Specification
   4. Software Product Specification

c. Descriptions of concept, designs, tests and delivered software
   1. Operational Concept Description
   2. System/Subsystem Design Description
   3. Interface Design Description
   4. Software Design Description
   5. Database Design Description
   6. Software Test Description
   7. Software Version Description

d. Manuals for users and support personnel
   1. Software User Manual
   2. Software Center Operator Manual
   3. Software Input/Output Manual
   4. Computer Operation Manual
   5. Computer Programming Manual
   6. Firmware Support Manual

e. Software Test Report to record, report and explain the results obtained from software testing

The standard stresses the use of natural software products, not additional documents. For this reason the products listed above are titled as descriptions, not documents, to remove the temptation to only consider them as traditional documents. The standard encourages alternative mediums for records such as data within computer-aided software engineering tools, and substitution by commercial manuals where applicable.

Whilst MIL-STD-498 makes the statement that it "invokes no other standards," it leaves specific details for some of the software-product content to related standardization documents such as ANSI/IEEE Std 1008 Standard for Software Unit Testing, or requires the developers to create and follow their own standards, such as "standards for representing requirements, design, code, test cases, test procedures, and test results."

3. MIL-STD-498 and Safety Requirements

Safety is regarded as one of three specific "critical requirements" along with security and privacy. However, safety considerations are left very short on detail about how to satisfy them and make no distinctions for varying degrees of safety level. The treatment of safety within MIL-STD-498 is largely limited to the section §4.2.4.1.

> 4.2.4.1 Safety assurance. The developer shall identify as safety-critical those CSCIs or portions thereof whose failure could lead to a hazardous system state (one that could result in unintended death, injury, loss of property, or environmental harm). If there is such software, the developer shall develop a safety assurance strategy, including both tests and analyses, to assure that the requirements, design, implementation, and operating procedures for the identified software minimize or eliminate the potential for hazardous conditions. The strategy shall include a software safety program, which shall be integrated with the system safety program if one exists. The developer shall record the strategy in the software development plan, implement the strategy, and produce evidence, as part of required software products, that the safety assurance strategy has been carried out.

§4.2.4.1 is a simple statement of what is required without providing any guidance to achieve it or issues to be considered when developing the required strategy and plan. The standard lists external standards to supplement the requirements of MIL-STD-498, these being MIL-STD-882 System Safety Program Requirements, MIL-HDBK-272 Safety Design and Evaluation Criteria for Nuclear Weapons Systems and IEEE Std 1228 Standard for Software Safety Plans. Treatment of safety in the various MIL-STD-498 DID is little better and is usually limited to precautionary notices, or specifying that safety requirements should be singled out “for special treatment” in separate subparagraphs. This is again done without any detailed requirements, guidance or considerations as to what special treatment entails. The meager offering in §3.7 of the Software Requirements Specification DID (extract included in Appendix A) is as detailed as the requirements for safety get in the standard or any of its DID.

4. MIL-STD-498 Software Product Evaluation and Quality Assurance

Section 5.15 describes the evaluation processes for software products. It distinguishes between in-process evaluations to be conducted by the developer, and evaluations associated with formal deliverables. The standard states requirements for the independence of evaluations and the retention of records. Appendix D of MIL-STD-498 provides fourteen evaluation criteria for the twenty-nine software products that it identifies. The evaluation criteria include the following types of considerations:

a. Contains all the applicable information of the relevant DID

b. Meets the Statement of Work and/or Contract Deliverables Requirements List if applicable

c. Is understandable (by the target audience)

d. Is internally consistent within a product

e. Was developed IAW the software development plan

f. Consistent with requirements at the system and software level

g. Are feasible to implement

h. Covers requirements, design, implementation etc.

In-process software Testing (unit testing, unit integration and testing, Computer Software Configuration Item (CSCI)/Hardware Configuration Item integration and testing) do not receive the same treatment as formal qualification of CSCI and system testing. In-process testing does not require testing personnel independence from development personnel and does not provide any guidance for evaluation criteria beyond the term ‘adequate.’ The ADF Airworthiness Design Requirements Manual has the following statement regarding the adequacy of MIL-STD-498.

> While many of the objectives under RTCA/DO-178B have placeholders in MIL-STD-498, there are no criteria that can be used to assess the adequacy of completion of the activity. For example, there is a requirement for unit and integration testing, but there are no criteria that

> define when testing can be considered complete. Therefore MIL-STD-498, in isolation, does not provide an adequate basis for software assurance and, by itself, is not recognized by the TAR as a software assurance standard. [14]

Software Quality Assurance of process and product is covered in section 5.16 of MIL-STD-498 (extract included in Appendix A). The purpose of this process is to ensure that activities are carried out; that the respective software products are produced and evaluated; and that identified problems are recorded, analyzed and corrected or justified. The standard also requires that, quality assurance activities be conducted by personnel that are independent of the development and product evaluation activities, and that quality assurance records be generated and retained.

III. SOFTWARE EVOLUTION, CERTIFICATION AND THE RELATIONSHIP WITH SOFTWARE STANDARDS

A. SOFTWARE EVOLUTION AS REENGINEERING

A useful representation of the steps involved during maintenance or modernization has been proposed by Kazman, Woods and Carriere [24]. The model in Figure 1 shows the multiple paths that software evolution may take from legacy code to new code.

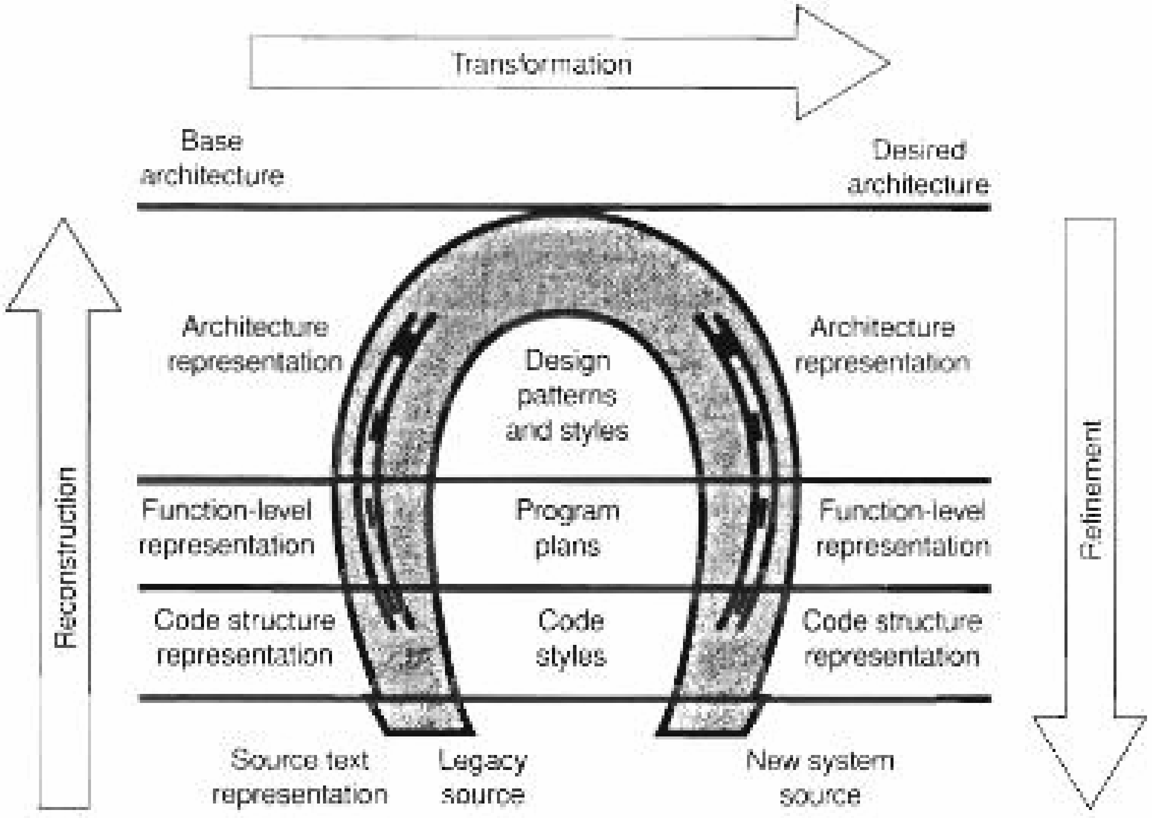
Transformation
Base architecture
Desired architecture
Reconstruction
Refinement
Architecture representation
Design patterns and styles
Architecture representation
Function-level representation
Program plans
Function-level representation
Code structure representation
Code styles
Code structure representation
Source text representation
Legacy source
New system source

Figure 1. Model for Software Reengineering (From: [1])

The horseshoe model reveals a vertical path up the left side for understanding the software by reconstruction from less to more abstract software products; lateral paths across the model for code, functional and architectural transformations; and a vertical path down the right side for refinement from software abstractions to code. It is not always necessary or desirable to travel the full path around the outside of the horseshoe. This would only be necessary if a major modernization effort was being undertaken and the only software product available from the legacy software development is the code, thus forcing a reengineering project. At the other end of the software evolution spectrum is minor modification. This may be achieved by a direct code transformation without attempting to reconstruct design and architecture.

In all cases of software evolution, the developer makes a practical choice of starting point on the left, reconstructs only as much as is necessary (if any), then makes the transformation at that level, finally proceeding through the refinement process to the new code (possibly only a partial refinement intended to simply incorporate new requirements).

B. SOFTWARE PRODUCT VERIFICATION

There is debate within the software engineering community over the merits or otherwise of inferring the quality of fielded software code from the quality of the people, processes, and tools that an organization uses to develop software products. It is generally accepted that whilst quality processes are a necessary enabler to produce quality code, processes alone are not a guarantee of quality code. It is for this reason, that the bulk of software processes for safety-critical software are in fact the software product verification processes that include a wide range of activities from requirements verification through unit testing to final qualification testing. Figure 2 [25] shows a taxonomy of the major types of verification activities that must be conducted on various software products. The testing branch of the verification techniques can be broken down further into the following sub-types:

1. Requirements-based testing that addresses the system and software functional and non-functional requirements
2. Function-based testing that addresses the software design functions
3. Structure-based testing that addresses the software implementation
4. Data-based testing that addresses different categories of data inputs, e.g. random inputs, equivalent partitions, normal inputs, abnormal inputs and boundary values
5. State-based testing for state-based software
6. Probability-based testing to assess software reliability
7. Fault-injection testing that uses prior experience to target likely sources of error, tests fault-tolerance performance, or tests the test.

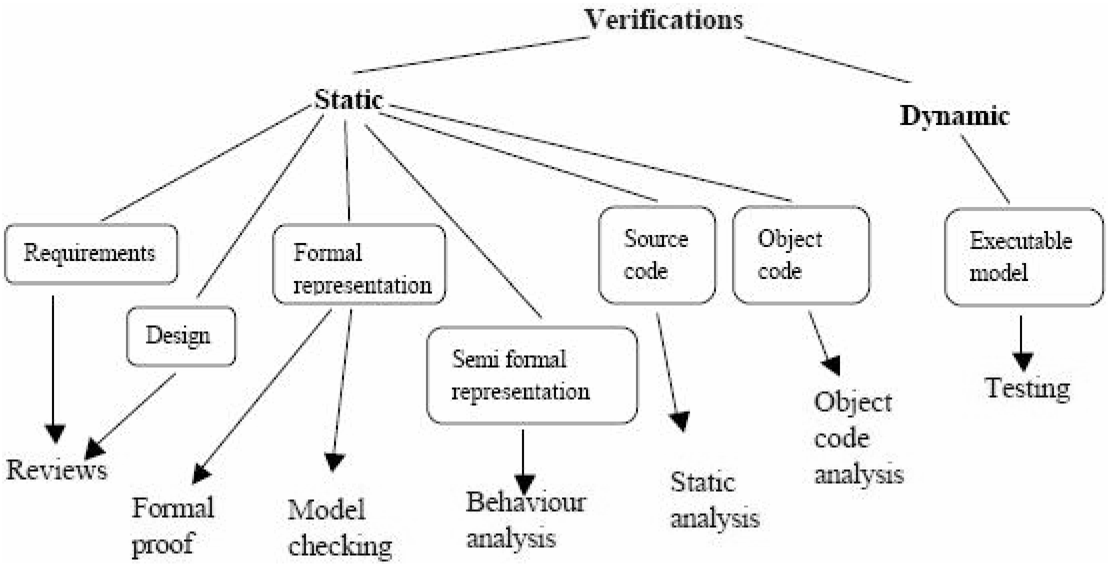

Figure 2 Verification Techniques (From: [25])

Given the practical impossibility of performing exhaustive testing, an important consideration for the verification of safety-critical software is the measure of coverage that is provided by the verification effort. The quantity and type of verification that is performed on software will determine the level of assurance that can be ascribed to the fielded product.

C. SOFTWARE EVOLUTION AND CERTIFICATION

Recertification must be sought when significant hardware or software changes are made to an aircraft design. MIL-HDBK-514 Operational Safety, Suitability, & Effectiveness for the Aeronautical Enterprise [26] provide the following instances that warrant recertification of airworthiness:

1. Changes that affect propulsion/drive system operation (including software)
2. Significant software revisions
3. Modification to weapons release/firing system, including stores management system and associated weapons system software

1. Using Deductive Logical for Software Certification

Software certification is based on a conclusion that software meets its quality specifications. In this paper, the quality specification of interest is safety; there are of course other quality objectives that exist, for instance security, which have their own certifications requirements. Accepting that the absolute definition of safety is not achievable for present-day complex systems, the practical conclusion that is drawn is that a software product meets all[4] of the software assurance objectives of an acceptable software assurance standard. A conclusion should only be accepted as valid when it flows naturally from the premises upon which it is founded.

The conclusion about software certification may be either positive (accepted) or negative (rejected) and in either case is based on one of the following implications:

a. If *all* of the software assurance objectives *are* satisfied, then the software is certifiable IAW the accepted software assurance standard, or

b. If *any* of the software assurance objectives *are not* satisfied, then the software is not certifiable IAW the accepted software assurance standard.

What remains to be determined for this basis of a conclusion is a technique to establish which of the above propositions is true. The following is a two-part proposal to achieve certification of evolved software:

a. Acceptance or rejection of evidence that was produced during the prior development of the software that is to be modified, and

b. Acceptance or rejection of evidence that is newly produced during the development of the evolved software.

It is understandable that not all of the evidence presented from prior development will be accepted as meeting the objectives of a contemporary software assurance standard and in some cases will not even be applicable. In the cases where the prior evidence is either inadequate or non-existent, new evidence will be required. New evidence must also be produced for the parts of the software development that are unique to the modification of the software. The focus of this thesis is a framework for determining the adequacy of the existing software products in support of airworthiness certification of evolving software.

2. Establishing the Safety Level Assessment

One of the early steps to be undertaken when commencing modification of safety-critical software is to either establish or revise the safety level for the modified software. A valid assessment of safety level is important as it determines the activities that need to

[4] Excluding *waivers* which are left as an issue for certification authorities that deal with them on a case-by-case basis.

be conducted and the evidence that will need to be presented to an airworthiness authority to achieve certification. Assessing the safety level can be achieved in the first instance, by examining the existing system/software safety program outputs from the previous development. When doing so, the assumptions made and analysis conducted for the previous development must be reviewed in the light of the proposed modifications, and then hazard identification and analysis must be conducted for the new requirements that are proposed. If a system/software safety program was not conducted for the previous development, or the information is insufficient or unavailable for any reason, a complete hazard identification and analysis will need to be conducted. The final outcome of this effort is the assessment of the failure condition category and requisite safety level for the modified software.

3. Carrying Out Software Evolution

Once the software level has been determined, development should proceed in a manner that facilitates the granting of an airworthiness certification. What this means is that sufficient evidence must be produced throughout the life-cycle that demonstrates the successful completion of the software engineering activities that address the objectives of software assurance.

4. Achieving Airworthiness Certification

The final step to achieving airworthiness certification is the collation of the evidence that supports certification. Two well known formats for this are the software accomplishment summary described in DO-178B or the safety case described in DEF STAN 00-55.

D. SOFTWARE EVOLUTION AND STANDARDS

The long time-frames over which aerospace systems are developed and then continue to evolve expose them to ongoing changes in software engineering. This evolution of the discipline of software engineering is manifest in the new computing technologies, and the design, implementation and verification techniques that are developed to cope with the increasing complexity of modern software-intensive systems. Another source of software engineering evolution that directly affects the standards domain is military acquisition reforms to reduce the number of military standards; efforts in this area aim to cancel rewrite or replace military standards with acceptable commercial-equivalent standards.

Given the cost to develop safety-critical software for aerospace applications, a reasonable expectation during software evolution is to be able to reuse the products that were produced during the previous software development. The challenge to achieving cost- and time-effective software reuse and the subsequent certification of the system containing the reused software is to identify the relationship between the activities and outputs of the previous development, and the activities and outputs required for the modification. This thesis presents a model for identifying the necessary relationships to facilitate software reuse in support of the certification of evolving safety-critical software.

The first version of software, version X in Figure 3, has a relationship R_1, to standard S_1, that meets or exceeds the requirements for certification IAW standard S_1 at the time of development of X. Software products must comply with multiple standards if no one standard provides all of the requirements needed for a software development project. Relationship R_1 explicitly represents just the one standard of interest; for the purposes of this thesis the type of standard of interest is that of software assurance.

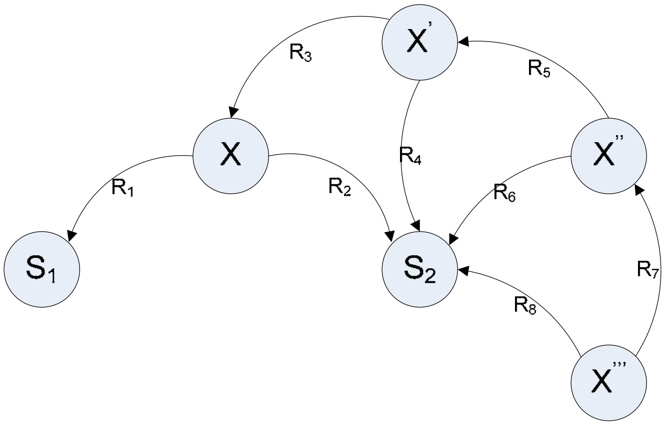

Figure 3. Software Evolution and Standards Relationship

The final set of software products that is produced during software development provides a complete definition of software version X in Figure 3. However, it is only a subset of this final set that is necessary to satisfy the requirements of standard S_1. The compliment of this subset contains the additional products sometimes regarded as internal to the development organization but which are necessary enablers for the activities that complete the software development.

Version X of the software also has a relationship R_2, to the contemporary standard that is now preferred, but either was not available, or was not applied at the time of previous development. This relationship exists from the moment that both standard S_2 and version X of the software both exist, but may be of little practical interest until the requirements of standard S_2 are invoked for version X of the software.[5]

Version X′ of the software has a relationship to version X (R_3), which is the result of modification due to maintenance or modernization. Some of the reasons for this modernization may be the addition of new features, removal of existing features, retargeting the software to new hardware or revamping of the interface. Version X′ of the software also has a relationship R_4, to standard S_2, which is the focus of this thesis. In order to achieve certification of version X′, relationship R_4 must satisfy the requirements of standard S_2 to the satisfaction of the certification authority. Versions X″ and X‴ and relationships R_5 through R_8 are further iterations of the same principle for subsequent modifications of the software. This model can also be applied to the introduction of a third software assurance standard, in which case second standard (formerly S_2) would be represented by S_1, and the third standard represented by S_2. Furthermore, the standards represented by S_1 and S_2 could be a different version of the old standard, for instance DO-178B and the pending DO-178C. Using this construction of the relationships, the evolution of the software product is represented by Equation 1:

$$X^n = X \bigcup_{s=1}^{n} \left(\Delta X_t \right)$$

Equation 1. Software Evolution

[5] Relationship R_2 may never be invoked and in fact is unlikely to be invoked if version X does not undergo any further changes.

ΔX_t represents the modification introduced at each iteration of the software product, such as

$$\begin{aligned} X''' &= X'' \bigcup \Delta X_3 \\ &= X' \bigcup \Delta X_2 \bigcup \Delta X_3 \\ &= X \bigcup \Delta X_1 \bigcup \Delta X_2 \bigcup \Delta X_3 \end{aligned}$$

Further development of an expression for software evolution in terms of the individual products to compose any given version requires an expression for software engineering that produces a single software version. A description of such an expression follows next.

E. ABSTRACT ALGEBRA AND MORPHISMS

Before applying the concepts of abstract algebra to software engineering, we briefly review the theory of abstract algebra in its familiar mathematical domain. In mathematics, all algebras are defined by a set of symbols A, and the set of operations Ω, that can be applied to the elements of A. Two algebras are said to be similar if they have the same number of operations for each arity. A purely hypothetical case would be two algebras are considered to be similar if they both have three unary operations, seven binary operations and two ternary operations. Furthermore, two algebras are said to be homomorphic if there is a function that provides a one-to-one correspondence between every element of one symbol and operation set $[\mathrm{A}_1, \Omega_1]$ and another symbol and operation set $[\mathrm{A}_2, \Omega_2]$.

$$\begin{aligned} &\text{If } \phi\big(\omega(a_1, a_2, ..., a_n)\big) = \omega'\big(\phi(a_1), \phi(a_2), ..., \phi(a_n)\big) \\ &\quad \text{for every } \omega \in \Omega, \\ &\quad \text{for every corresponding } \omega' \in \Omega', \\ &\quad \text{for every } a_i \in \mathrm{A} \\ &\quad \text{where } n \text{ is defined by the arity of } \omega \\ &\text{Then } \phi \text{ is a homomorphism from } [\mathrm{A}, \Omega] \text{ to } [\mathrm{A}', \Omega'] \end{aligned}$$

Definition of Homomorphism

To illustrate this with a common example, consider the homomorphic function of logarithm that provides such a correspondence between multiplication and addition and the set of symbols (numbers) that are valid for logarithms. One algebra *R*, is defined by the symbol set of all positive real numbers and the single binary operation of multiplication, that is $R = \left[\mathbb{R}^{+}, \{\times\}\right]$. A second algebra *L*, is defined by the symbol set of all real numbers (negative and positive) and the single binary operation of addition, that is $L = \left[\mathbb{R}, \{+\}\right]$. The reader might be tempted to propose that there are many other operations that could be performed on the elements $\mathbb{R}^{+}$ and $\mathbb{R}$, but those operations would be outside the definition presented here, and constitute a different algebra than either *R* or *L*. Note also, that there is no requirement for the symbol sets A_1 and A_2 to be equivalent. The two algebras R and L in this example are similar by virtue of each having only one binary operation. Furthermore, algebras *R* and *L* are said to be homomorphic by reason of the function $\phi(x) = \log(x)$. For example:

$$\begin{aligned}
&\text{Using } \phi\left(\omega\left(a_1, a_2, \dots, a_n\right)\right) = \omega'\left(\phi\left(a_1\right), \phi\left(a_2\right), \dots, \phi\left(a_n\right)\right) \\
&\quad \text{substituting } \phi = Log \\
&Log\left(\omega\left(a_1, a_2\right)\right) = \omega'\left(Log\left(a_1\right), Log\left(a_2\right)\right) \\
&\quad \text{substituting } \omega = \times, \text{ and } \omega' = + \\
&Log\left(\times\left(a_1, a_2\right)\right) = +\left(Log\left(a_1\right), Log\left(a_2\right)\right) \\
&\quad \text{hence} \\
&Log\left(a_1 \times a_2\right) = Log\left(a_1\right) + Log\left(a_2\right)
\end{aligned}$$

Equation 2. Homomorphism Example – Logarithms

Another example of a homomorphic function is the Laplace Transform that permits convolution in the linear time domain to be represented as multiplication in the frequency domain.

F. ABSTRACT ALGEBRA AND SOFTWARE DEVELOPMENT

The relationships in Figure 3 are composed of the *products* (symbol set) that are used and produced during the *activities* (operations) that are conducted during software evolution. Before applying the mathematical concept of homomorphism to software evolution and standards interoperability, we first specify the sets A and Ω as follows:

1. A represents the set of all possible software products (α_r) both used and created during software development, and
2. Ω represents the set of all possible activities (ω_s) that are conducted on the software products during software development.

Some obvious examples of the products α_r, in set A are the Plans, Requirements, Infrastructure, Design, Source Code, Executable Code, Verification results. Section 1.5 of the IEEE Standard for Software Reviews [27] has a list of thirty-seven 'software products' with the inclusion of such items as 'anomaly reports,' 'build procedures,' 'installation procedures,' and 'walkthrough reports' in addition to the previously mentioned products. Examples of the activities ω_s, in set Ω are Planning, Development, Verification, Configuration Management, Quality Assurance, Certification etc. for the various stages of software development.

Using the algebraic representation for software development we can now write:

$$\omega_s\left(B_s\right)\rightarrow\alpha_t$$

where conducting activity ω_s,
on an appropriate subset B_s,
that has been produced prior to activity ω_s,
produces a new software artifact α_t.

Equation 3. Single Product Development

Equation 4 provides an example of this algebraic description as the development of code:

$$\text{Software Coding}\begin{pmatrix}\text{Development Plan}\\ \text{Coding Standards}\\ \text{Development Tools}\\ \text{Design}\end{pmatrix}\rightarrow\text{Code}$$

Equation 4. Single Product Development – Code Example

Each of the products used in the software coding activity (which in this example is a quaternary operation) are themselves products of earlier software development activities, that is, planning, defining standards, choosing and qualifying tools and developing the software design. Development tools also encompass configuration management and traceability systems in addition to the obvious tools such as text editors and compilers. Design includes the software architecture and design-level software requirements.[6]

While the source and object code are being produced by the software coding activity, other important products are also being generated. These additional products include updates to traceability information and configuration management data in addition to the creation of feedback for the design and requirements activities in case any errors (e.g., conflicting requirements) are found in them as a consequence of carrying out the software coding activity. Including these products into the representation expands the previous description of software development to that of Equation 5:

$$\omega_s\left(B_s\right) \rightarrow \left(C_s\right)$$

where conducting activity ω_s,
on an appropriate subset B_s,
that has been produced prior to activity ω_s,
produces the new subset of the software artifacts C_s,
such that $\left(B_s \subset C_s\right)$.

Equation 5. Single Activity

Equation 5 specifies that the set of products that are produced (C_S) by an activity is a proper superset of the products used (B_S) during the activity. This would always be the case, even in situations were some portion of the requirements, design or implementation is removed. At the very least, the removed product should remain as part of the development history for the software.

[6] As distinct from system or high-level software requirements.

The source code development example presented in Equation 4 is expanded to have the following relationships that demonstrate the production of multiple software products from a single software development activity:

$$\text{Software Coding}\left(\begin{array}{c}\text{Development Plan}\\ \text{Coding Standards}\\ \text{Development Tools}\\ \text{Design}\end{array}\right)\begin{array}{l}\rightarrow \text{Code}\\ \rightarrow \text{Traceability Data}\\ \rightarrow \text{CM Data}\\ \rightarrow \text{Trouble Reports}\end{array}$$

Equation 6. Single Activity – Coding Example

The last step in the algebraic representation of software development is to represent the composition of all the development activities that together produce the completed software package.

$$\bigcup_{s=1}^{m} \omega_s\left(B_s\right) \rightarrow \mathrm{A}_1$$

where the union of m activities ω_s,
on the appropriate subsets of artifacts B_s,
produces the final set of artifacts A_1.

Equation 7. Software Development

The last step in the algebraic representation of software development is to represent the composition of all the development activities that together produce the completed software package.

G. ABSTRACT ALGEBRA AND SOFTWARE EVOLUTION

Combining the representations in Equation 1 and Equation 7 provides the composite expression in Equation 8 for the total output of software development as a product evolves.

$$\text{For} \quad X^n = X \bigcup_{s=1}^{n} \left(\Delta X_t \right)$$

$$\text{where } X = \bigcup_{t=1}^{p} \omega_t \left(B_t \right)$$

$$\text{and} \quad \Delta X = \bigcup_{t=1}^{q_n} \omega_t \left(B_t \right)$$

$$\text{then} \quad X^n = \bigcup_{t=1}^{p} \omega_t \left(B_t \right) \bigcup_{s=1}^{n} \left(\bigcup_{t=1}^{q_n} \omega_t \left(B_t \right) \right)$$

Equation 8. Software Evolution

It is desirable from a cost-benefit perspective that some of the products that were produced during previous software development be reused to achieve certification during software evolution. In terms of the model presented in this thesis, this amounts to how much of the software product package X^n satisfies the relationship $R_{2(n+1)}$. The greater the amount of previously developed products that can be reused is clearly an important business objective, but this must be achieved within the bounds of an acceptable level of software assurance to meet airworthiness requirements.

In terms of the first iteration of software evolution, how much of the development for version $X^{'}$, is available as reused software product from the development of version X, can be legitimately used to satisfy the relationship R_4? A promising answer to this problem lies in how much of the development for version X, that was used to satisfy relationship R_1 could have been used to satisfy the relationship R_2? The next section of this chapter examines this question.

H. APPLICATION OF MORPHISM TO SOFTWARE EVOLUTION

The mathematical concept of homomorphism was introduced in §III.E in order to use it now as the basis for describing the correspondence of the products that constitute the relationships R_1 and R_2. A function that transforms the products developed for version X and satisfies relationship R_1, to products that satisfy relationship R_2 can be used to identify the amount of reuse of existing software product that is available for certification of the evolved software. Unfortunately, the definition of homomorphism has a single function that provides the mapping between two sets of similar algebra. It would indeed be nice if a single function existed to map all the products satisfying relationship

R_1 to also satisfy relationship R_2. It would be perfect if such a function was an identity function and hence no additional operations or activities were required to achieve the desired correspondence. This is highly unlikely given that standard S_2 either did not exist or was not chosen as the basis for certification of the previous software development. Three likely scenarios for morphims to the products that satisfy relationship R_1 to make them acceptable for relationship R_2 are:

1. Some of the existing products will be able to be mapped using an identity function, in which case no additional activity is required for them to be considered as meeting the requirements of S_2,

$$\phi(\alpha_t) \rightarrow \alpha_t$$

Equation 9. Identity Morphism

2. Some of the existing products will be able to be mapped using a partial function, in which case some rework or additional activity is required for these to be used to meet the requirements of S_2,

$$\phi(\alpha_t) \rightarrow \delta\alpha_t$$

Equation 10. Partial Morphism

3. Some of the existing products will not be able to be mapped using any function; or rather these products will be subject to a null function in the transformation. In these cases, completely new work and software products will be required to satisfy the requirements of relationship R_2.

$$\phi(\alpha_t) \rightarrow \varnothing$$

Equation 11. Null Morphism

Achieving effective reuse during airworthiness certification is a matter of maximizing the amount of products that can be found in the first two of these groups. Ensuring adequate airworthiness is a matter of ensuring that products that are in the last two groups are not mistakenly included in the first group.

This concept of morphism is applied to the software coding activity in the following sections as an example for code generation. The next section has an underlying premise that certification-related aspects to code generation that are important for safety-critical software are ensuring that software code and requirements are fully traceable in both directions, and that code has been verified as correct and complete. These aspects are described in the following paragraphs and presented in the $\omega_s\left(B_s\right) \rightarrow \left(C_s\right)$ algebraic notation.

1. Previous Software Development

The activity of software coding has plans, standards, tools and design as inputs to this process. The chief product that is produced by the coding process (at least with respect to an executable product) is code. Other products that are produced by the software coding activity are traceability data, configuration management data and trouble reports. These are not necessary to produce an executable product and may be regarded by the code-centric developer as by-products of producing executable code, but they are essential to achieving the software assurance necessary for a certifiable product. Equation 6 is presented again below for the sake of completeness in this section.

$$\text{Software Coding}_{\text{X}}\begin{pmatrix}\text{Development Plan}_{\text{X}} \\ \text{Coding Standards} \\ \text{Development Tools} \\ \text{Design}_{\text{X}}\end{pmatrix}\begin{matrix}\rightarrow \text{Code}_{\text{X}} \\ \rightarrow \text{Traceability Data}_{\text{X}} \\ \rightarrow \text{CM Data}_{\text{X}} \\ \rightarrow \text{Trouble Reports}_{\text{X}}\end{matrix}$$

Equation 6. Single Activity – Coding Example

For safety-critical software, it is necessary to verify that all low-level requirements have been met and to also ensure that there is no additional code that does not have a traceable link back to requirements. Establishing traceability between code and design (i.e., low-level requirements) is a process that has as inputs to the process; code, design for that code, and development tools that permit traceability to be recorded. The output traceability data is added to the traceability database of the software project.

$$\text{Code Traceability}_{\text{X}}\begin{pmatrix}\text{Design}_{\text{X}} \\ \text{Code}_{\text{X}} \\ \text{Development Tools}\end{pmatrix} \rightarrow \text{Traceability Data}_{\text{X}}$$

Equation 12. Code Traceability

The purpose of software testing is to ensure that the software design has been implemented and does not contain errors. The inputs to this process are the code itself and the verification tools that enable testing. (For the sake of simplicity in this example, other inputs such as test scripts are considered to be part of the verification tools.) The outputs from the testing process are the obvious test results, but also include traceability and configuration management data and trouble reports.

$$\text{Software Testing}_X \begin{pmatrix} \text{Code}_X \\ \text{Test Infrastructure} \\ \text{Test Plans, Cases and Procedures}_X \\ \text{Test Input Data}_X \end{pmatrix} \begin{matrix} \rightarrow \text{Test Logs and Results}_X \\ \rightarrow \text{Traceability Data}_X \\ \rightarrow \text{CM Data}_X \\ \rightarrow \text{Trouble Reports}_X \end{matrix}$$

Equation 13. Software Testing

This completes the description of the previous development of software products associated with the software coding and the additional activities necessary for certification of this small element of software development. The next section describes the software coding and related activities for modification of the legacy software.

2. Software Modification

Plans, standards, tools and design are once again the inputs to the software coding activity. However some of the activities undertaken and artifacts used during modification may not be the same as those for the legacy software development. The plans and design used for software modification will likely be different to that used for the previous development, while the coding standards and development tools may or may not be the same as previously used. In this example, it is assumed that the coding standards and development tools are in fact the same as that used for the previous development. Furthermore, no distinction is drawn between the deliberate reuse of standards and tools and the unplanned reuse of the same that would be characterized as salvage rather than reuse. New trouble reports, updated traceability data and configuration data will again be generated in addition to the code as a consequence of the modification.

As the software coding activity is being conducted for the purposes of maintenance or modernization and not replacement, the actual code produced during the Software Coding$_M$ activity is likely to be just a fraction of the total code that defines the final product. These considerations are expressly represented in Equation 14. In addition to the modified code, the code that is unchanged from the previous development must retain its traceability, correctness and completeness during the modification. The final modified code is described by Equation 15.

$$\text{Software Coding}_{\text{M}}\begin{pmatrix}\text{Development Plan}_{\text{M}}\\ \text{Coding Standards}\\ \text{Development Tools}\\ \text{Design}_{\text{M}}\end{pmatrix}\begin{matrix}\to \text{Code}_{\text{M}}\\ \to \text{Traceability Data}_{\text{M}}\\ \to \text{CM Data}_{\text{M}}\\ \to \text{Trouble Reports}_{\text{M}}\end{matrix}$$

Equation 14. Software Coding – Modification

$$\text{Code} = \left(\text{Code}_{\text{O}}\right)\cup\left(\text{Code}_{\text{M}}\right)$$

Equation 15. Software Code Evolution

Carrying out the activity of code traceability is also going to be a process that considers both modified and unaffected software products. In the same manner as for code generation, traceability data will be described by the union of outputs from both pre- and post- modification traceability products described by Equation 16 and Equation 17.

$$\text{Code Traceability}_{\text{M}}\begin{pmatrix}\text{Design}_{\text{M}}\\ \text{Code}_{\text{M}}\\ \text{Development Tools}\end{pmatrix}\to \text{Traceability Data}_{\text{M}}$$

Equation 16. Code Traceability – Modification

$$\text{Traceability Data} = \left(\text{Traceability Data}_{\text{X}}\right)\cup\left(\text{Traceability Data}_{\text{M}}\right)$$

Equation 17. Software Traceability Evolution

The same rationale applies to the software testing activity conducted for the final modified software and the description of configuration management data, trouble reports and test results that are produced by the activity. These are represented by Equation 18 through Equation 21.

$$\text{Software Testing}_{\text{M}}\begin{pmatrix} \text{Code}_{\text{M}} \\ \text{Test Infrastructure} \\ \text{Test Plans, Cases and Procedures}_{\text{M}} \\ \text{Test Input Data}_{\text{M}} \end{pmatrix}\begin{matrix} \to \text{Test Logs and Results}_{\text{M}} \\ \to \text{Traceability Data}_{\text{M}} \\ \to \text{CM Data}_{\text{M}} \\ \to \text{Trouble Reports}_{\text{M}} \end{matrix}$$

Equation 18. Software Testing – Modification

$$\text{Test Logs and Results} = \left(\text{Test Logs and Results}_{\text{X}}\right) \cup \left(\text{Test Logs and Results}_{\text{M}}\right)$$

Equation 19. Software Test Results Evolution

$$\text{CM Data} = \left(\text{CM Data}_{\text{X}}\right) \cup \left(\text{CM Data}_{\text{M}}\right)$$

Equation 20. Software Configuration Management (CM) Data Evolution

$$\text{Trouble Reports} = \left(\text{Trouble Reports}_{\text{X}}\right) \cup \left(\text{Trouble Reports}_{\text{M}}\right)$$

Equation 21. Software Trouble Reports Evolution

3. Software Product Morphism

The previous two sections described a portion of the software development (software coding and related activities); first for previous software coding, and then for the subsequent modification of the code. This section addresses which of the software products from this total effort of previous development and modification might be used to satisfy software assurance certification of the final modified software. Some of the products from the previous development may be satisfactory, even if the currently desired standard was not met for the previous development. It is likely however that at least some of the software product will not be acceptable for the current certification effort, even if the software assurance standard has not changed, this being the more likely scenario if a new software assurance standard has been imposed upon the modification.

a. Code Traceability

We firstly deal with traceability between design and code, which we have asserted is one of the certification requirements associated with the software coding activity in order to meet a software assurance standard. Equation 17 shows the two components that comprise the full code traceability for the final modified code; existing and modification traceability data. Acceptance that the software modifications were carried out with the preplanned intention to meet a new software assurance standard, leads to a reasonable expectation that the traceability data generated for the modified segments of code will be acceptable. Traceability between unmodified code and design may not be properly established after modification, especially if the traceability prior to modification is questionable; this element of traceability is explored further in the following paragraphs.

Table 3 shows a simplified example that demonstrates some of the variations to code traceability as a consequence of modifications to code. Columns one and two represent traceability between design elements and code units of the previous development, while columns one and three represent traceability between design elements and code units of the final modified software. The implementation of design element 'A' is shown as traceable to code unit 'M' (and vice-versa) and remains unchanged by the modification. The implementation of design element 'B' has been restructured by the modification and is now traced to the new code unit 'R' in addition to the previous code unit 'N'. Design element 'C' was implemented in code unit 'O' of the previous code, but has now been removed as a result of the modification. Traceability information for design element 'D' is not shown for either the previous or the modified code. The intention in this hypothetical example is that design element 'D' does exist as a necessary element of the design and is present in the software (as verified by requirements-based testing), but its traceability information was simply overlooked in the previous development. Also for the purposes of this example, design element 'D' was not associated with or considered during the software modification which prevents anything being said about its traceability after the modification. Code unit 'Q' is included in Table 3 to represent code that does not have any identified backwards traceability to design. Again, this reappears in the modified software because it was not specifically addressed as part of the modification. Design element ‘E’ is a new feature that is introduced as part of the modification and is traced to code unit ‘S’.

Design Element	Previous Code Unit	Modified Code Unit
A	M	M
B	N	N R
C	O	▨
D	?	?
?	Q	Q
E	▨	S

Table 3. Modified Code Traceability

Consider the final traceability data that is proposed to meet certification requirements in the two parts identified in Equation 17; firstly the later of the two parts, that which was generated during the modification, and lastly the earlier of the two parts, that which was generated for the previous development.

$$\text{Traceability Data} = \left(\text{Traceability Data}_{\text{X}}\right) \cup \left(\text{Traceability Data}_{\text{M}}\right)$$

Equation 17. Software Traceability Evolution

In this example, the software modification was composed of three facets: (i) refactoring of design element ‘B’, (ii) removal of design element ‘C’ and (iii) addition of design element ‘E’. It would be expected that the traceability data for design element ‘E’ will be properly detailed as part of the modification effort and as such, satisfy the new certification requirements. We propose that the new traceability for design element ‘B’ will also be complete after the modification, although this will be somewhat different from the traceability of design element ‘B’ for the previous development. Traceability for design element ‘C’ will not exist, nor should it, for the modified software.

The traceability data from the previous development (Traceability Data$_X$ in Equation 17), even in this simple scenario, requires a morphism that is a composition of identity, partial and null morphisms. Consider each of the design elements and code units in turn.

The traceability data for design element ‘A’ for the previous development can probably be used as-is for the modified software. In this case the identity morphism applies:

$$\phi\left(\text{Traceabilty}_{\text{A}_\text{X}}\right) \rightarrow \text{Traceabilty}_{\text{A}_\text{M}}$$

Equation 22. Identity Morphism – Code Traceability

The software developer would very likely use the previous design element ‘B’ traceability data as a basis for establishing code traceability in the modified code. If this is the case, the final traceability of design element ‘B’ will be a composite of the new traceability data to code unit ‘R’ with a partial morphism of the previous traceability data. This is represented in Equation 23.

$$\text{Traceability Data}_\text{B} = \left(\delta\left(\text{Traceabilty}_{\text{B}_\text{X}}\right) \rightarrow \text{Traceabilty}_{\text{B}_\text{N}}\right) \cup \left(\text{Traceability Data}_{\text{B}_\text{R}}\right)$$

Equation 23. Partial Morphism – Design Element ‘B’ Code Traceability

The traceability data for design element ‘C’ for the previous development cannot be used for the modified software. In this example, the design element and corresponding code unit have been removed from the modified software. Traceability data for this portion of the software in the previous development is no longer applicable to the final modified code. Furthermore, as it is not part of the modification (by omission), any latent reference to it in the final traceability data will be an error. In this case the null morphism should apply as in Equation 24:

$$\phi\left(\text{Traceabilty}_{\text{C}_\text{X}}\right) \rightarrow \varnothing$$

Equation 24. Null Morphism – Code Traceability

Design element ‘D’ and code unit ‘Q’ do not have any identified traceability data from previous development and hence nothing can be said with certainty about their traceability after the software modification. There is no traceability information for them and therefore nothing to subject to a morphism function.

This example only focuses on a narrow aspect of traceability between design and code. Traceability between other software products is also needed. It is usual practice to have all of the traceability information collected into one traceability matrix or database. In this case, the reader would appreciate that identification of existing traceability data for appropriate reuse becomes an increasingly complex problem with variation in the amount of partial morphism that would need to be applied to different software products.

b. Software Testing

The design elements and code units already presented in Table 3 are used again for the following analysis of test results for the modified code. We again treat the latter of the two parts of the complete test results first, before discussing the reuse of previous development test results.

$$\text{Test Logs and Results} = \left(\text{Test Logs and Results}_{\text{X}}\right) \cup \left(\text{Test Logs and Results}_{\text{M}}\right)$$

Equation 19. Software Test Results Evolution

We again assume that the testing of modified code satisfies the requirements for the new software assurance standard (and any unmodified code that may interface the modification). This assumption equates to full acceptance of code unit R and S test logs and results, and partial acceptance or possibly full acceptance of unit M test logs and results. What remains is to determine the suitability of the test results for the previous development that have not been covered as part of the modification. *N.B.*: We assume here that the certification requirements placed on testing of the evolved software has changed and that this new standard is more stringent in its requirements.

The degree to which test plans, cases, procedures, input data, logs and results associated with code unit N from the previous development can be reused will depend on the manner in which testing was conducted for code units N and R after modification of the software. We believe it to be likely that test cases, descriptions and procedures would be completely redeveloped for the modification in all but the most trivial modification efforts. This belief is based on the fact that design element B has been refactored across two code units and the effort required to plan and conduct completely new tests would be justified by comparison to the effort required to develop only some new tests but integrate them with reused existing tests and results. This assumption is highly dependent on the amount of refactoring, but we believe that refactoring to include a second code unit would be significant enough to justify the assumption; under this assumption we propose that the morphism to be applied to existing test results for code unit N would be a null morphism. Testing for code unit O, like traceability for it, has no bearing on the modified software and should also be subjected to the null morphism.

$$\phi\left(\text{Test Logs and Results}_{\mathrm{N_X}}\right) \rightarrow \varnothing$$

Equation 25 Null Morphism –Test Logs and Results

As was stated in the previous discussion regarding code traceability, the code for design element D has been tested, despite not having traceability data to identify which code unit provides the implementation. Test results could be obtained without the code unit identification by undertaking requirements-based integration testing. However, as mentioned in §III.B, there are other testing techniques in addition to requirements-based testing that we propose would be the requirements of the newly applied software assurance standard S_2. In this case the mapping of test logs and results for design element D will be the partial morphism.

$$\delta\left(\text{Test Logs and Results}_{\mathrm{X}}\right) \rightarrow \text{Test Logs and Results}_{\mathrm{M_D}}$$

Equation 26. Partial Morphism – Design Element ‘D’ Test Logs and Results

Once again, code unit ‘Q’ does not have any identified design element. Code unit Q is thus unlikely to have any test logs and results from the previous development and so again, there is nothing to subject to a morphism function.

The preceding discussion pertaining to the reuse of prior development test logs and results only covered the issue of existence of the software products, not the suitability of the products that do exist. An assessment of the adequacy of the previous test logs and results to meet the new standard would have to be made. The measure of adequacy would include what class (timing/performance/loading etc.) of testing was conducted, what coverage was provided (requirements/statement/state/range -based), and the evaluation criteria. It is likely that under these considerations, the degree of partial morphism .for many, if not all, previous test logs and results would be further reduced by a partial morphism.

c. General Considerations for Software Product Morphism

The preceding discussion has not distinguished whether the assigned safety level of the modified software has changed from the assigned safety level for the previous development. In the cases where the assigned safety level has been increased, the amount of software product from the previous development that can be reused for the purposes of certification of the modified software would be expected to be significantly reduced. It would be probable that any of the software products to be reused would be subject to partial morphism and require additional work to make them suitable for use in the new certification application.

IV. RELATED WORK

A. ARCHITECTURAL TRANSFORMATION

Grunske's work on the semi-automatic improvement of the nonfunctional properties of software using hypergraph transformations [28] is founded on the concept of morphisms to modify architectural elements during the design stage of software development. This work compliments the approach proposed in this thesis by providing a technique to assess a given system's architecture for its ability to achieve specified requirements for software quality such as system safety. The technique provides an assessment of whether the software architecture complies with stated requirements for software quality, or requires alteration to meet system requirements.

It is cost effective to continually conduct evaluations of the nonfunctional properties of a system from the earliest possible opportunity. Grunske proposes a method for semi-automatically conducting a comparison of nonfunctional properties of a system with different architectures of the required components. The comparison begins with an architectural specification that satisfies the entire set of functional requirements, and then continues with alternative architectural elements that are available in the analysis tool. If the evaluation determines that a nonfunctional requirement is unachievable to the desired level, such as safety, then the architectural specification must be transformed to one that achieves the quality performance requirements without compromising the attainment of functional requirements. If no impediments to nonfunctional properties are identified, then the development process can proceed using the originally proposed architectural specification.

1. Hierarchical Typed Hypergraphs

The work is based on Hierarchical Typed Hypergraph (HTH) theory. *Hypergraph* theory is an extension of graph theory that permits more than two nodes (vertices) to an edge. *Typed hypergraphs* are a restriction within general hypergraphs that stipulates node and hyperedge *types*; hyperedges of a certain type may only connect nodes of certain types. *Hierarchical Typed Hypergraphs* are typed hypergraphs that use hyperedges called *complex hyperedges* that are themselves hypergraphs. This permits a recursive construction of a full HTH using a lesser HTH, hyperedges and nodes.

A HTH is characterized by the tuple $\langle V, E, att, lab, ext, cts \rangle$ where:

V is a finite set of nodes (vertices) from the set of node types L_V,
E is a finite set of hyperedges from the set of hyperedge types L_E,
att is the attachment function to assign a sequence of nodes to a hyperedge,
lab is the labeling function for nodes and hyperedges,
ext describes a sequence of external nodes, and
cts is an assignment function that specifies that a hyperedge contains a HTH.

2. Unified Modeling Language for Real-Time

Grunske applies his theory to architectures specified using the Unified Modeling Language for Real-Time (UML-RT). To do so, he derives the UML-RT metaclass *capsule* from the HTH metaclass *hypergraph*, the UML-RT metaclass *port* (both end and relay *types*) to the HTH metaclass *node*, and the UML-RT metaclass *connector* to the HTH metaclass *hyperedge*. Figure 4 presents these relationships.

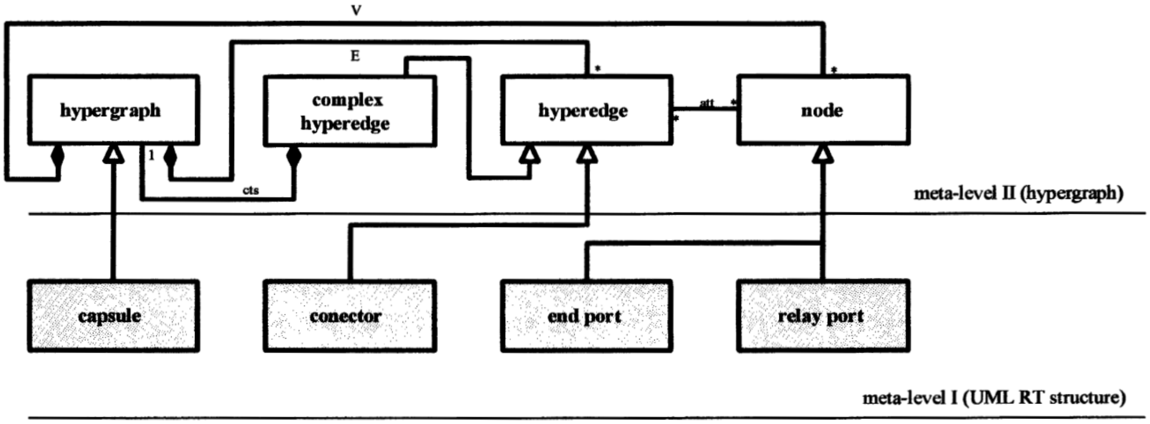

Figure 4. Metaclass Mapping of HTH and UML-RT Elements (From: [28])

3. Hierarchical Typed Hypergraphs Transformations

Hypergraph transformation is achieved by identifying sub-hypergraphs within hypergraphs to enable hypergraph subtraction, hypergraph addition to construct an expanded hypergraph, and hypergraph morphisms of attachments between nodes and hyperedges to complete the reconstruction. A HTH morphism (*m*) from one HTH (*G*) to a functionally equivalent HTH (*G'*), uses pairs of mappings for nodes and hyperedges. For instance, $m: G \rightarrow G' = < m_V, m_E >$ where $m_V : V \rightarrow V'$ and $m_E : E \rightarrow E'$.

The graphical T-notation represented in Figure 5 is used to identify the operands of the transformation as follows:

1. Ports (nodes) and connections (hyperedges) in the lower left corner of the T are to be subtracted from the original HTH.
2. Ports and connections (and in the case of Figure 5, new additional components) in the lower right corner of the T are to be added to the HTH.
3. Ports above the T are retained in the architectural transformation.
4. Morphisms of the attachments between ports and connections reconfigure the resultant architecture to retain functional equivalence with the original architecture.

The example presented in Figure 5 shows a single capsule on the left being replaced with a subsystem that has three capsules[7] on the right, that provide their output to a voting capsule that chooses which of the messages to use based on a two-out-of-three voting system.

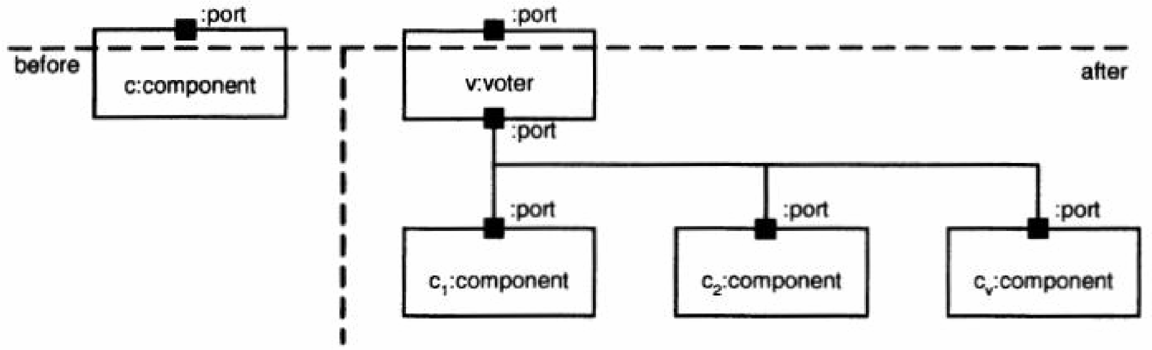

Figure 5. Hypergraph Graphical T-Notation (From: [28])

[7] Functionally equivalent to the original subtracted capsule in this example, but that need not be the case.

4. Evaluation of Quality Characteristics

Each element of an architectural specification is annotated with the relevant aspects of the quality characteristics of interest and the capsules extended with analysis models to facilitate the architectural evaluation. For example, to evaluate reliability or safety, the elements will be annotated with failure data (safe and unsafe), and extended with a fault tree analysis model.

Architectural transformations can then step through a pre-defined library of possible transformations and be evaluated for performance against the nonfunctional properties of concern.

B. COMPUTER-AIDED SOFTWARE EVOLUTION

Harn [29] extends the use of hypergraphs to a relational hypergraph model as a means for formaly defining software evolution. This was done by relating *software evolution objects* as inputs and outputs of the software *steps* that use and produce the objects. A hypergraph capturing this relationship is defined in [29] as the tuple $\langle N, E, I, O \rangle$ where:

N is a set of nodes representing software evolution objects,

E is a set of hyperedges representing steps during software evolution,

I is the set of inputs for each hyperedge, and

O is the set of outputs from each hyperedge.

In a relational hypergraph, every input node on a hyperedge is either a *primary*, or a *secondary* input to the hyperedge, and the output of the step is *dependent* on all input nodes. Primary-input-driven hypergraphs relate different *versions* of the same software evolution object, while secondary-input-driven hypergraphs relate different *software evolution objects*. Figure 6 shows an example of a relational hypergraph where the output software product (P2.2) is produced as a new (and merged) version of earlier software products (P1.1, $P_1$2.2 and $P_2$2.2) and is also dependent on another software product (R1.1).

> A primary-input-driven path addresses the evolution history of a software evolution component based on the change from an old version to a new version. A secondary-input-driven path addresses the evolution rationale with a sequence of the software evolution components. Therefore, these two structures form the relational hypergraph which determines not only what to evolve but also how to evolve it. [29]

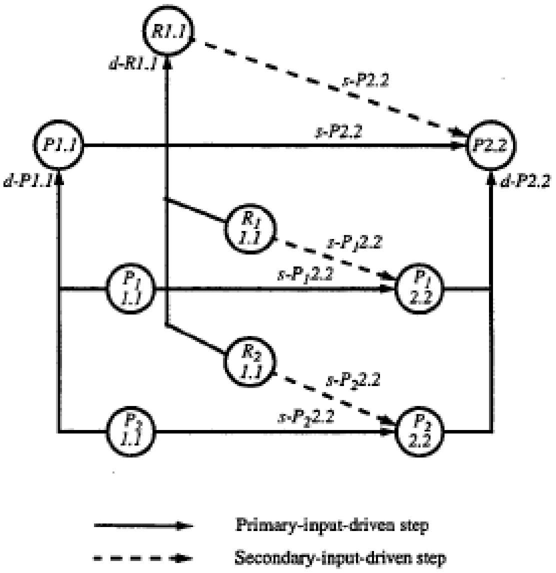

Figure 6. Relational Hypergraph (From: [29])

The functional notation using abstract algebra proposed in this thesis has the following similarities to the relational hypergraph model presented in Harn's dissertation:

1. The use of the function operator for software development activities presented in Equation 5 is similar to the use of hyperedges as steps of software evolution.
2. Software products being identified as the operands of development activities is equivalent to nodes representing input software evolution objects to software evolution steps.
3. Software product being produced as the outputs from development activities equate nodes representing software evolution objects as outputs from steps.

Harn's work contrasts the work of this thesis, which is a technique for the analysis of the suitability of previously developed software product for reuse, when the software and the standards being applied are both evolving. As previously mentioned, this reuse may be intentional or be regarded as salvage when the reuse of the software product is not preplanned.

The work in [29] covers the deliberate reuse of software evolution objects during software development. Although Harn does not mention software assurance certification as one of the steps of software evolution, the tool that was developed could be extended to incorporate certification.

C. F/RF-111C AGM-142E/1760 INTEGRATION

The AGM-142E/1760 Integration as part of the F/RF-111C Block C4 SCS upgrade was initially thought to be candidate for application of the technique proposed in this thesis. Unfortunately, the software for the System Interface Processor (SIP) is not an example of legacy safety-critical software that was having its certification basis upgraded during modification.

AGM-142E/1760 Integration involves both the modification of software for several processors[8] already embedded in the F/RF-111C and incorporation of a SIP. The SIP provides the necessary interface between the MC and AGM-142E loaded onto aircraft Weapon Stations (WS) which was not possible through the existing SMP interface to the WS. The certification basis for the existing subsystems was to remain a tailored version of MIL-STD-498, while DO-178B (software level B) was proposed as the certification basis for the SIP software. Figure 7 is a simplified block diagram showing the data and control connections between system components post integration.

The software units within the SIP are the operating system, bus driver, discrete input/output driver, analog to digital converter driver which were COTS products and the OFP which was a completely new software development.

[8] Mission Computer (MC), Stores Management Processor (SMP) Bus Sub-System Integration Unit, and System Function Processor.

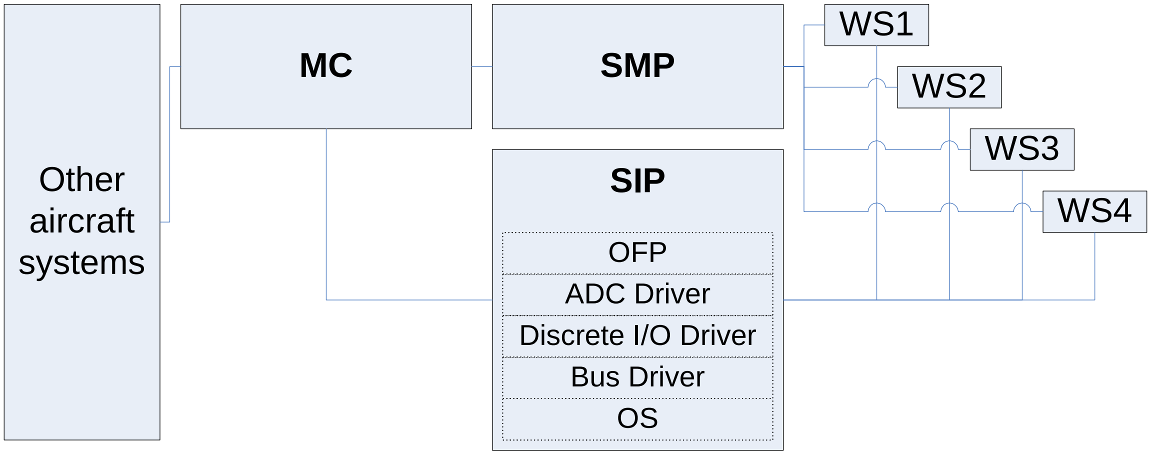

Figure 7. AGM-142E Integration – Simplified Block Diagram

Certifying COTS components as part of a DO-178B certified system is a related but distinct problem from that of upgrading the certification basis for previously developed software. Certification of COTS products is well recognized within DO-178B and guidance to achieve the desired certification is included within the standard.

The possibility of applying abstract algebra to describe the development of the OFP existed but was not explored. The reason for not investigating this project further was that the OFP was a new development and hence no software products existed from a previous development to which the proposed morphisms could be applied.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSION

A. KEY FINDINGS AND ACCOMPLISHMENTS

In this thesis we investigated the ADF practice of certifying modified legacy software to DO-178B where the modification is developed in compliance with DO-178B, but where the software was previously developed to some other standard. The ADF practice is based on an extension of the FAA policy for certification of software to DO-178B, which has been previously developed to an earlier version of the DO-178 standard, and subsequently modified in such a manner as to be compliant with DO-178B.

The original scope of the research was to examine the objectives of DO-178B and make a comparison with another software assurance or development standard to see how much of the previous software development activities and products could possibly comply with the certification requirements of DO-178B. If sufficient software products and activities are found to be satisfactory, then some form of automation of any technique to reuse the software products from a previous development would be necessary to make such reuse practical. A necessary first step in achieving automation of any software product reuse is to define a mathematical description of software evolution to enable the mapping of applicable software activities and products between software standards. Thus, deriving such a mathematical description of software development and evolution became the primary focus of this work.

In this thesis, we introduced an algebraic model for software development as a means to apply a systematic approach to define software development and evolution. Our algebraic model introduces the application of abstract algebra to software development by defining software *products* as the *set of symbols*, and software development *activities* as the *set of operations*.

The abstract algebra model has a function notation that defines software development activities. This function notation was developed for a single activity, software development, and software evolution.

$$\omega_s\left(B_s\right)\rightarrow\left(C_s\right)$$

Equation 5. Single Activity

$$\bigcup_{s=1}^{m}\omega_s\left(B_s\right)\rightarrow \mathrm{A}_1$$

Equation 7. Software Development

$$X^n=\bigcup_{t=1}^{p}\omega_t\left(B_t\right)\bigcup_{s=1}^{n}\left(\bigcup_{t=1}^{q_n}\omega_t\left(B_t\right)\right)$$

Equation 8. Software Evolution

To assist in the efficient achievement of recertification against new or evolving standards, we demonstrated the application of morphism to map software products for reuse during software evolution. Three general functions that were presented are (i) the identity function for software product that can be reused without additional effort, (ii) the partial function for software product that is only partially reusable and (iii) the null function for software product that cannot be reused to achieve recertification.

$$\phi\left(\alpha_t\right)\rightarrow\alpha_t$$

Equation 9. Identity Morphism

$$\phi\left(\alpha_t\right)\rightarrow\delta\alpha_t$$

Equation 10. Partial Morphism

$$\phi\left(\alpha_t\right)\rightarrow\varnothing$$

Equation 11. Null Morphism

Whilst the model presented in this thesis is yet to be completed, and specific morphisms for mapping software products are still to be identified, the approach has been well received and shows promise of satisfying the basis for automation of recertification.

B. CONCLUDING REMARKS

A comment raised towards the end of this work was in essence a question of "does it really matter once a system is in service?" Mapping prior activities and software products to a new software assurance standard does not, on its own, make a safety-critical system any safer: to make software-intensive systems safer, the code must be improved. This is a valid assertion but the reply to this point is that software safety assurance is an attempt to obtain a measure of the *confidence* in the safety aspects of a software-intensive system.

Establishing the current level of software safety assurance based on existing available evidence permits program managers to make justifiable decisions concerning the continued operation, or otherwise, of aircraft with or without software upgrades. The first step in the decision-making process is to determine the current level of software safety assurance. If the finding gives the certification authority sufficient grounds for claiming an acceptable level of system safety has been achieved, then the system can continue to be operated without any further expenditure or effort. This prevents unwarranted expenditure of resources on software evolution that is not necessary from the software safety certification perspective. If the finding does not satisfy the certification authority, then the following options can be explored to reach a decision. The first option is always *do nothing* and continue to operate the system as-is. This sounds unpalatable when talking about safety, but is none-the-less one of the options. A decision to take this option can either be an educated or uneducated one; determining the current level of software safety assurance gives the decision maker the opportunity to make an educated decision. The second option is to modify the software within the system to raise the software safety assurance level. The third option is to discontinue operation of the existing system and replace it. These last two choices are also better made with a clear understanding of a system's existing level of software assurance. We hope that the technique proposed in this thesis will be effective for obtaining evidence from the existing software products, upon which an educated choice about continued system operation or evolution can be made.

C. FUTURE WORK

1. Determine the Morphisms Required for Activities and Products

The immediate work to be continued is the identification of a complete set of morphisms needed to map the requirements of a legacy standard such as MIL-STD-498 to the objectives of the desired standard DO-178B. Completeness of the set of morphisms will be achieved when the requirements of the desired standard that *can* be achieved with the legacy software products *are* identified by appropriate morphisms.

2. Case Studies

A candidate for investigation and application of the proposed technique is the AF/A-18 software development being undertaken by the RAAF at the Tactical Fighter Weapon System Support Unit. The MC and SMP OFP for the AF/A-18 is an active area of the evolution of legacy software which was originally developed to MIL-STD-498. The aircraft will be subject to continued software maintenance and modernization and has sufficient remaining useful life to warrant the investigation into upgrading the software certification basis.

Other possibilities within the ADF might be the RAAF F/RF-111C MC and SMP software development which currently have a MIL-STD-498 certification basis; or software support for the Royal Australian Navy's Super Seasprite helicopters, FFG Frigates or Collins class submarines.

3. Automation

We feel that the model and notation for the representation of software development, evolution and recognition of prior software product presented in this thesis is amenable to automation through the development of extensions to existing software development tools.

4. Different Combinations of Other Standards

The original focus of this thesis was a comparative analysis of the DO-178B objectives and MIL-STD-498 requirements to determine the amount of MIL-STD-498 compliant software product that would likely be acceptable for certification of software to DO-178B.

5. Application to Other Domains and Dimensions

The methodology presented in this thesis can be applied to other domains that have safety-critical concerns. These include rail system automation, nuclear power plant control and medical devices.

Another dimension of high assurance software that might benefit from the application of this technique is software security certification.

It is possible that the approach introduced here may also be applied to other engineering disciplines that are concerned with recertification.

6. Cost-Benefit Analysis With Respect to Software/System Age

Achieving certification to a new standard is a considerable effort with significant costs. It is important therefore, to obtain an acceptable ROI. One could explore whether the cost of recertification would diminish as a system gets closer to the end of its service life. Cost-benefit analysis of recertification of a system as a function of the remaining useful life of the system is another possibility for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX: EXTRACTS FROM SELECTED STANDARDS

A. EXTRACTS FROM RTCA DO-178B

12.0 ADDITIONAL CONSIDERATIONS

12.1 Use of Previously Developed Software

The guidelines of this subsection discuss the issues associated with the use of previously developed software, including assessment of modifications, the effect of changing an aircraft installation, application environment, or development environment, upgrading a development baseline, and SCM and SQA considerations. The intention to use previously developed software is stated in the Plan for Software Aspects of Certification.

12.1.1 Modifications to Previously Developed Software

This guidance discusses modifications to previously developed software where the outputs of the previous software life cycle processes comply with this document. Modification may result from requirement changes, the detection of errors, and/or software enhancements. Analysis activities for proposed modifications include:

a. The revised outputs of the system safety assessment process should be reviewed considering the proposed modifications.

b. If the software level is revised, the guidelines of paragraph 12.1.4 should be considered.

c. But the impact of the software requirements changes and the impact of software architecture changes should be analyzed, including the consequences of software requirement changes upon other requirements and coupling between several software components that may result in reverification effort involving more than the modified area.

d. The area affected by change should be determined. This may be done by data flow analysis, control flow analysis, timing analysis and traceability analysis.

e. Areas affected by the change should be reverified considering the guidelines of section 6.

12.1.4 Upgrading a Development Baseline

Guidelines follow for software whose software life cycle data from a previous application are determined to be inadequate or do not satisfy the objectives of this document, due to the safety objectives associated with a new application. These guidelines are intended to aid in the acceptance of:

- Commercial off-the-shelf software.
- Airborne software developed to other guidelines.
- Airborne software developed prior to the existence of this document.
- Software previously developed to this document at a lower software level

Guidance for upgrading a development baseline includes:

a. The objectives of this document should be satisfied while taking advantage of software lifecycle data of the previous development that satisfy the objectives for the new application.

b. Software aspects of certification should be based on the failure conditions and software level(s) as determined by the system safety assessment process. Comparison to failure conditions of the previous application will determine areas which may need to be upgraded.

c. Software life cycle data from a previous development should be evaluated to ensure that the software verification process objectives of the software level are satisfied for the new application.

d. Reverse engineering may be used to regenerate software life cycle data that is inadequate or missing in satisfying the objectives of this document. In addition to producing the software product, additional activities may need to be performed to satisfy the software verification process objectives.

e. If use of product service history is planned to satisfy the objectives of this document in upgrading a development baseline, the guidelines of paragraph 12.3.5 should be considered.

f. The applicant should specify the strategy for accomplishing compliance with this document in the Plan for Software Aspects of Certification.

12.1.5 Software Configuration Management Considerations

If previously developed software is used, the software configuration management process for the new application should include, in addition to the guidelines of section 7:

a. Traceability from the software product and software life cycle data of the previous application to the new application.

b. Change control that enables problem reporting, problem resolution, and tracking of changes to software components used in more than one application.

12.1.6 Software Quality Assurance Considerations

If previously developed software is used, the software quality assurance process for the new application should include, in addition to the guidelines of section 8:

a. Assurance that the software components satisfy or exceed the software life cycle criteria of the software level for the new application.

b. Assurance that changes to the software life cycle processes are stated in the software plans.

12.3.5 Product Service History

If equivalent safety for the software can be demonstrated by the use of the software's product service history, some certification credit may be granted. The acceptability of this method is dependent on:

- Configuration management of the software.
- Effectiveness of problems reporting activity.
- Stability and maturity of the software.
- Relevance of product service history environment.
- Actual error rates and product service history.
- Impact of modifications.

Guidance for the use of product service history includes:

a. The applicant should show that the software and associated evidence used to comply with system safety objectives have been under configuration management throughout the product service history.

b. The applicant should show that the problem reporting during the product service history provides assurance that representative data is available and that in-service problems were reported and recorded, and are retrievable.

c. Configuration changes during the product service history should be identified and the effect analyzed to confirm the stability and maturity of the software. Uncontrolled changes to the Executable Object Code during the product service history may invalidate the use of product service history.

d. The intended software usage should be analyzed to show the relevance of the product service history.

e. If the operating environments of the existing and proposed applications differ, additional software verification should confirm compliance with the system safety objectives.

f. The analysis of configuration changes and product service history environment may require the use of software requirements and designed data to confirm the applicability of the product service history environment.

g. If the software is a subset of the software that was active during the service period, Then analysis should confirm the equivalency of the new environment with the previous environment, and determine those software components that were not executed during normal operation

Note: Additional verification may be needed to confirm compliance with the system safety objectives for those components.

h. The problem report history should be analyzed to determine how safety-related problems occurred and which problems were corrected.

i. Those problems that are indicative of an inadequate process, such as design or code errors, should be indicated separately from those whose cause are outside the scope of this document, such as hardware or system requirements errors.

j. The data described above and these items should be specified in the Plan for Software Aspects of Certification:

(1) Analysis of the relevance of the product service history environment.

(2) Length of service period and rationale for calculating the number of hours in the service, including factors such as operational modes, the number of independently operating copies in the installation and in service, and the definition of " normal operation" and " normal operation time".

(3) Definition of what was counted as an error and rationale for that definition.

(4) Proposed acceptable error rates and rationale for the product service history period in relation to the system safety and proposed error rates.

k. If the error rate is greater than that identified in the plan, these errors should be analyzed and the analyses reviewed with the certification authority.

ANNEX A (of DO-178B)

PROCESS OBJECTIVES AND OUTPUTS BY SOFTWARE LEVEL

Objective			Applicability by S/W Level				Output		CC by S/W Level			
	Description	Ref.	A	B	C	D	Description	Ref.	A	B	C	D
Software Planning Process												
1	Software development and integral processes activities are defined.	4.1a 4.3	○	○	○	○	PSAC SDP SVP SCMP SQAP	11.1 11.2 11.3 11.4 11.5	1 1 1 1 1	1 1 1 1 1	1 2 2 2 2	1 2 2 2 2
2	Transition criteria, inter-relationships and sequencing among processes are defined.	4.1b 4.3	○	○	○							
3	Software life cycle environment is defined.	4.1c	○	○	○							
4	Additional considerations are addressed.	4.1d	○	○	○	○						
5	Software development standards are defined.	4.1e	○	○	○		S/W Requirements Stds S/W Design Stds S/W Code Stds	11.2 11.3 11.4	1 1 1	1 1 1	2 2 2	
6	Software plans comply with RTCA DO-178B.	4.1f 4.6	○	○	○		SQA Records S/W Verification Results	11.19 11.14	2 2	2 2	2 2	
7	Software plans are coordinated.	4.1g 4.6	○	○	○		SQA Records S/W Verification Results	11.19 11.14	2 2	2 2	2 2	
Software Development Processes												
8	High-level requirements are developed.	5.1.1a	○	○	○	○	S/W Requirements Data	11.9	1	1	1	1
9	Derived high-level requirements are defined.	5.1.1b	○	○	○	○	S/W Requirements Data	11.9	1	1	1	1
10	Software architecture is developed.	5.2.1a	○	○	○	○	Design Description	11.10	1	1	2	2
11	Low-level requirements are developed.	5.2.1a	○	○	○	○	Design Description	11.10	1	1	2	2
12	Derived low-level requirements are developed.	5.2.1b	○	○	○	○	Design Description	11.10	1	1	2	2

Objective			Applic-ability by S/W Level				Output		CC by S/W Level			
	Description	Ref.	A	B	C	D	Description	Ref.	A	B	C	D
13	Source Code is developed.	5.3.1a	○	○	○	○	Source Code	11.11	1	1	1	1
14	Executable Object Code is produced and integrated in the target computer.	5.4.1a	○	○	○	○	Executable Object Code	11.12	1	1	1	1
Verification of Outputs of Software Requirements Process												
15	Software high-level requirements comply with system requirements.	6.3.1a	●	●	○	○	S/W Verification Results	11.14	2	2	2	2
16	High-level requirements are accurate and consistent.	6.3.1b	●	●	○	○	S/W Verification Results	11.14	2	2	2	2
17	High-level requirements are compatible with target computer.	6.3.1c	○	○			S/W Verification Results	11.14	2	2		
18	High-level requirements of verifiable.	6.3.1d	○	○	○		S/W Verification Results	11.14	2	2	2	
19	High-level requirements conform to standards.	6.3.1e	○	○	○		S/W Verification Results	11.14	2	2	2	
20	High-level requirements are traceable to system requirements.	6.3.1f	○	○	○	○	S/W Verification Results	11.14	2	2	2	2
21	Algorithms are accurate.	6.3.1g	●	●	○		S/W Verification Results	11.14	2	2	2	
Verification of Outputs of Software Design Process												
22	Low-level requirements comply with high-level requirements.	6.3.2a	●	●	○		S/W Verification Results	11.14	2	2	2	
23	Lower-level requirements are accurate and consistent.	6.3.2b	●	●	○		S/W Verification Results	11.14	2	2	2	
24	Low-level requirements are compatible with target computer	6.3.2c	○	○			S/W Verification Results	11.14	2	2		
25	Level-level requirements are verifiable.	6.3.2d	○	○			S/W Verification Results	11.14	2	2		
26	Level-level requirements conform to standards.	6.3.2e	○	○	○		S/W Verification Results	11.14	2	2	2	

Objective			Applic-ability by S/W Level				Output		CC by S/W Level			
	Description	Ref.	A	B	C	D	Description	Ref.	A	B	C	D
27	Level-level requirements are traceable to high-level requirements.	6.3.2f	○	○	○		S/W Verification Results	11.14	2	2	2	
28	Algorithms are accurate.	6.3.2g	●	●	○		S/W Verification Results	11.14	2	2	2	
29	Software architecture is compatible with high-level requirements.	6.3.3a	●	○	○		S/W Verification Results	11.14	2	2	2	
30	Software architecture is consistent.	6.3.3b	●	○	○		S/W Verification Results	11.14	2	2	2	
31	Software architecture is compatible with target computer.	6.3.3c	○	○			S/W Verification Results	11.14	2	2		
32	Software architecture is verifiable.	6.3.3d	○	○			S/W Verification Results	11.14	2	2		
33	Software architecture conforms to standards.	6.3.3e	○	○	○		S/W Verification Results	11.14	2	2	2	
34	Software partitioning integrity is confirmed.	6.3.3f	●	○	○	○	S/W Verification Results	11.14	2	2	2	2
Verification of Outputs of Software Coding & Integration Processes												
35	Source Code complies with low-level requirements.	6.3.4a	●	●	○		S/W Verification Results	11.14	2	2	2	
36	Source Code complies with software architecture.	6.3.4b	●	○	○		S/W Verification Results	11.14	2	2	2	
37	Source Code is verifiable.	6.3.4c	○	○			S/W Verification Results	11.14	2	2		
38	Source Code conforms to standards.	6.3.4d	○	○	○		S/W Verification Results	11.14	2	2	2	
39	Source Code is traceable to low-level requirements.	6.3.4e	○	○	○		S/W Verification Results	11.14	2	2	2	
40	Source Code is accurate and consistent.	6.3.4f	●	○	○		S/W Verification Results	11.14	2	2	2	
41	Output of integration process is complete and correct.	6.3.5	○	○	○		S/W Verification Results	11.14	2	2	2	

Objective			Applic-ability by S/W Level				Output		CC by S/W Level			
	Description	Ref.	A	B	C	D	Description	Ref.	A	B	C	D
Testing of Outputs of Integration Process												
42	Executable Object Code complies with high-level requirements.	6.4.2.1 6.4.3	○	○	○	○	S/W Verification Cases and Procedures S/W Verification Results	11.13 11.14	1 2	1 2	2 2	2 2
43	Executable Object Code is robust with high-level requirements.	6.4.2.2 6.4.3	○	○	○	○	S/W Verification Cases and Procedures S/W Verification Results	11.13 11.14	1 2	1 2	2 2	2 2
44	Executable Object Code complies with low-level requirements.	6.4.2.1 6.4.3	●	●	○		S/W Verification Cases and Procedures S/W Verification Results	11.13 11.14	1 2	1 2	2 2	
45	Executable Object Code is robust with low-level requirements.	6.4.2.2 6.4.3	●	○	○		S/W Verification Cases and Procedures S/W Verification Results	11.13 11.14	1 2	1 2	2 2	
46	Executable Object Code is compatible with target computer.	6.4.3a	○	○	○	○	S/W Verification Cases and Procedures S/W Verification Results	11.13 11.14	1 2	1 2	2 2	2 2
Verification of Verification Process Results												
47	Test procedures are correct.	6.3.6b	●	○	○		S/W Verification Results	11.14	2	2	2	
48	Test results are correct and discrepancies explained.	6.3.6c	●	○	○		S/W Verification Results	11.14	2	2	2	
49	Test coverage of high-level requirements is achieved.	6.4.4.1	●	○	○	○	S/W Verification Results	11.14	2	2	2	2
50	Test coverage of low-level requirements is achieved.	6.4.4.1	●	○	○		S/W Verification Results	11.14	2	2	2	
51	Test coverage of software structure (modified condition/decision) is achieved.	6.4.4.2	●				S/W Verification Results	11.14	2			
52	Test coverage of software structure (decision coverage) is achieved.	6.4.4.2a 6.4.4.2b	●	●			S/W Verification Results	11.14	2	2		
53	Test coverage of software structure (statement coverage) is achieved.	6.4.4.2a 6.4.4.2b	●	●	○		S/W Verification Results	11.14	2	2	2	

Objective			Applic-ability by S/W Level				Output		CC by S/W Level			
	Description	Ref.	A	B	C	D	Description	Ref.	A	B	C	D
54	Test coverage of software structure (data coupling and control coupling) is achieved.	6.4.4.2c	●	●	○		S/W Verification Results	11.14	2	2	2	
Software Configuration Management Process												
55	Configuration items are identified.	7.2.1	○	○	○	○	S/W Configuration Management Records	11.18	2	2	2	2
56	Baselines and trace ability are established.	7.2.2	○	○	○	○	S/W Configuration Index S/W Configuration Management Records	11.16 11.18	1 2	1 2	1 2	1 2
57	Problem reporting, change control, change review, and configuration status accounting and established.	7.2.3 7.2.4 7.2.5 7.2.6	○	○	○	○	Problem Reports S/W Configuration Management Records	11.17 11.18	2 2	2 2	2 2	2 2
58	Archive, retrieval, and release are established.	7.2.7	○	○	○	○	S/W Configuration Management Records	11.18	2	2	2	2
59	Software load control is established.	7.2.8	○	○	○	○	S/W Configuration Management Records	11.18	2	2	2	2
60	Software life cycle environment control is established.	7.2.9	○	○	○	○	S/W Life Cycle Environment Configuration Index S/W Configuration Management Records	11.15 11.18	1 2	1 2	1 2	2 2
Software Quality Assurance Process												
61	Assurance is obtained that software development and integral processes comply with approved software plans and standards.	8.1a	●	●	●	●	S/W Quality Assurance Records	11.19	2	2	2	2
62	Assurance is obtained that transition criteria for the software lifecycle processes are satisfied.	8.1b	●	●			S/W Quality Assurance Records	11.19	2	2		
63	Software conformity review is conducted.	8.1c 8.3	●	●	●	●	S/W Quality Assurance Records	11.19	2	2	2	2

Objective			Applic-ability by S/W Level				Output		CC by S/W Level			
	Description	Ref.	A	B	C	D	Description	Ref.	A	B	C	D
Certification Liaison Process												
64	Communication and understanding between the applicant and the certification authority is established.	9.0	○	○	○	○	PSAC	11.1	1	1	1	1
65	The means of compliance is proposed and agreement with the Plan for Software Aspects of Certification is obtained.	9.1	○	○	○	○	PSAC	11.1	1	1	1	1
66	Compliance substantiation is provided	9.2	○	○	○	○	Software Accomplishment Summary Software Configuration Index	11.20 11.16	1	1	1	1
Legend												
○	The objective should be satisfied.											
●	The objective should be satisfied with independence.											
Blank	Satisfaction of objective is at applicant’s discretion.											
1	Data satisfies the objectives of Control Category 1 (CC1).											
2	Data satisfies the objectives of Control Category 2 (CC2).											

Table 4. Objectives, Activities, Outputs and Data Control Categories (After: 15, Tables A-1 through A-10)

B. EXTRACTS FROM MIL-STD-498

4.2.2 Standards for software products. The developer shall develop and apply standards for representing requirements, design, code, test cases, test procedures, and test results. These standards shall be described in, or referenced from, the software development plan.

5.16 Software quality assurance. The developer shall perform software quality assurance in accordance with the following requirements.

Note: If a system or CSCI is developed in multiple builds, the activities and software products of each build should be evaluated in the context of the objectives established for that build. An activity or software product that meets those objectives can be considered satisfactory even though it is missing aspects designated for later builds. Planning for software quality assurance is included in software development planning (see 5.1.1).

5.16.1 Software quality assurance evaluations. The developer shall conduct on-going evaluations of software development activities and the resulting software products to:

a. Assure that each activity required by the contract or described in the software development plan is being performed in accordance with the contract and with the software development plan.

b. Assure that each software product required by this standard or by other contract provisions exists and has undergone software product evaluations, testing, and corrective action as required by this standard and by other contract provisions.

5.16.2 Software quality assurance records. The developer shall prepare and maintain records of each software quality assurance activity. These records shall be maintained for the life of the contract. Problems in software products under project-level or higher configuration control and problems in activities required by the contract or described in the software development plan shall be handled as described in 5.17 (Corrective action).

5.16.3 Independence in software quality assurance. The persons responsible for conducting software quality assurance evaluations shall not be the persons who developed the software product, performed the activity, or are responsible for the software product or activity. This does not preclude such persons from taking part in these evaluations. The persons responsible for assuring compliance with the contract shall have the resources, responsibility, authority, and organizational freedom to permit objective software quality assurance evaluations and to initiate and verify corrective actions.

C. EXTRACT FROM DATA ITEM DESCRIPTION DI-IPSC-81433 (SOFTWARE REQUIREMENTS SPECIFICATION)

3.7 Safety requirements. This paragraph shall specify the CSCI requirements, if any, concerned with preventing or minimizing unintended hazards to personnel, property, and the physical environment. Examples include safeguards the CSCI must provide to prevent inadvertent actions (such as accidentally issuing an "auto pilot off" command) and non-actions (such as failure to issue an intended "auto pilot off" command). This paragraph shall include the CSCI requirements, if any, regarding nuclear components of the system, including, as applicable, prevention of inadvertent detonation and compliance with nuclear safety rules.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

1 Seacord, R. C., Plakosh, D., and Lewis, G. A., Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices, Addison-Wesley, 2003.

2 Leveson, N. G., Safeware: System Safety and Computers, Addison-Wesley, 1995.

3 Boeing Integrated Defense Systems, F/A-18 Hornet Milestone, [http://www.boeing.com/defense-space/military/fa18/fa18_milestones.htm] February 2006.

4 Boeing Integrated Defense Systems, F/A-18 Background Info, [http://www.boeing.com/defense-space/military/fa18/fa18_4back.htm] February 2006.

5 Jane's, All the Worlds Aircraft, [http://www.janes.com] February 2006.

6 Jane's, Aircraft Upgrades, [http://www.janes.com] February 2006.

7 Defence Materiel Organisation, Project Air 5376 – F/A-18 Hornet Upgrade, [http://www.defence.gov.au/dmo/asd/air5376/air5376.cfm] February 2006.

8 Finnish Air Force, The Hornet's Ten Years in Service, [http://www.ilmavoimat.fi/index_en.php?id=651] February 2006.

9 Canada National Defence, Canada's Air Force, Aircraft: CF-18 Hornet – Future Plans, [http://www.airforce.forces.gc.ca/equip/cf-18/future_e.asp] February 2006.

10 Department of Defence, Media Invitation, [http://www.defence.gov.au/media/DepartmentalTpl.cfm?CurrentId=234] February 2006.

11 Boeing Commercial Airplanes, 747, [http://www.boeing.com/commercial/747family/index.html] February 2006.

12 Bowers, P., The F/A-18 Advanced Weapons Lab Successfully Delivers a $120-Million Software Block Upgrade, CrossTalk, pp. 10-11, January 2002.

13 Storey, N., Safety-Critical Computer Systems, Pearson Prentice Hall, 1996.

14 DAIRENG-DGTA. DI(AF) AAP 7001.054: Airworthiness Design Requirements Manual. S2Ch7P19. Defence Air Publications Agency. 23 February 2004.

15 Radio Technical Commission for Aeronautics, DO-178B, Software Considerations in Airborne Systems and Equipment Certification, 1 December 1992.

16 Department of Defense, MIL-STD-882D, Standard Practice for System Safety, 10 February 2000.

17 Roland, H. E. and Moriarty, B., System Safety Engineering and Management, John Wiley & Sons, 1990.

18 Institute of Electrical and Electronics Engineers, Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, 1990.

19 National Aeronautics and Space Administration, Standard NASA-STD-8739.8, Software Assurance Standard, 28 July 2004.

20 Wiktionary Contributors, Assurance [http://en.wiktionary.org/wiki/assurance]. February 2006.

21 Department of Defense, MIL-STD-882C, System Safety Program Requirements, 19 January 1993.

22 Society of Automotive Engineers, ARP4754, Certification Considerations for Highly-Integrated or Complex Aircraft Systems, 12 January 1996.

23 Johnson, L. A., DO-178B, Software Considerations in Airborne Systems and Equipment Certification, CrossTalk, web edition, October 1998.

24 Kazman, R., Woods, S.G., and Carriere, S.J., Requirements for Integrating Software Architecture and Reengineering Models: CORUM II, Proceedings of the Fifth Working Conference on Reverse Engineering, pp. 154-163, 1998.

25 Swedish Defence Materiel Administration. H ProgSäkE: Handbook for Software in Safety Critical Applications. FMV, Sweden. 2001.

26 Department of Defense, MIL-HDBK-514, Operational Safety, Suitability, & Effectiveness for the Aeronautical Enterprise, 28 March 2003.

27 Institute of Electrical and Electronics Engineers, Standard 1028-1997, IEEE Standard for Software Reviews, 1997.

28 Yang, H.. Advances in UML and XML-Based Software Evolution, Idea Group Inc., 2005.

29 Harn, M.-C., Computer-Aided Software Evolution Based on Inferred Dependencies, Ph.D. Dissertation, Naval Postgraduate School, Monterey, California, December 1999.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Professor Bret Michael
   Naval Postgraduate School
   Monterey, California

4. Dr. Jeff Voas
   SAIC, Inc.
   Arlington, Virginia

5. Professor George Dinolt
   Naval Postgraduate School
   Monterey, California

6. SQNLDR Benjamin Musial
   Royal Australian Air Force
   RAAF Williams, Laverton, Australia

7. Professor Duminda Wijesekera
   George Mason University
   Fairfax, Virginia

8. Dr. John Harauz
   Jonic Systems Engineering, Inc.
   Willowdale, Ontario, Canada

9. FLTLT Desmond Meacham
   Royal Australian Air Force
   RAAF Williams, Laverton, Australia